INFORMATION MODELING AND MEDIATION LANGUAGES AND TECHNIQUES FOR INFORMATION SHARING AMONG HETEROGENEOUS INFORMATION SYSTEMS

By

TSAE-FENG YU

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1997

To
My Parents,
My Wife,
and My Daughters, Emily and Christine

ACKNOWLEDGMENTS

I wish to express my immense gratitude to my advisor, Dr. Stanley Y.W. Su, for his excellent guidance and support in the past four years. Without his help, this research work could not have been done.

I would also like to express my gratitude to the cochairman of my committee, Dr. Herman Lam, for his assistance in revising my dissertation and invaluable suggestions on this research. I would like to thank Dr. Sharma Chakravarthy, Dr. Paul Fishwick, and Dr. Jose Principe for serving on my committee and providing feedback.

Many thanks are due to our overworked secretary, Ms. Sharon Grant, who provided great support and assistance on my research during this long period of time. I would like to thank my friends in the Database Center, especially Fang Wang, Sooha Lee, Javier Arroyo-Figueroa, Zhidong Yang, and Simon Jou for all their efforts in supporting my work. I would also like to thank all the students at the Database Systems Research and Development Center for being such great fellows.

Last, but not least, I want to express my gratitude to my wonderful wife, Yen Shiah, for her constant encouragement and inspiration on my research and for her efforts in making a great family.

This research is part of the National Industrial Information Infrastructure Protocols (NIIIP) project and is sponsored by the Advanced Research Projects Agency (ARPA), through an IBM grant (# 9M1BE4H), assigned to the University of Florida, Gainesville, Florida.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Dissertation
Presented to the Graduate School of the University of Florida
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

INFORMATION MODELING AND MEDIATION LANGUAGES AND TECHNIQUES FOR INFORMATION SHARING AMONG HETEROGENEOUS INFORMATION SYSTEMS

By

Tsae-Feng Yu

May 1997

Chairman: Dr. Stanley Y. W. Su
Cochairman: Dr. Herman Lam
Major Department: Computer and Information Science and Engineering

Information sharing among heterogeneous information systems has been a major challenge to the information processing community. Two major obstacles impeding its realization are *data* heterogeneity and *system* heterogeneity. The traditional schema integration approach cannot effectively resolve data heterogeneity problems because it forces all users to view their data in the same way, as defined by an integrated view. Furthermore, this approach is not scalable nor extensible when applied in a large-scale information system environment.

A promising approach to data heterogeneity problems is to perform run-time mediation to convert one data representation to another to suit each user's view of data. Despite some recent research efforts on information mediation, an effective and efficient mediation technique is still needed. Such a mediation technique must be scalable and extensible. It must also be compatible with the technique used to resolve system heterogeneous problems.

In this research, we focus on information mediation in the context of global query processing. We introduce an extensible common modeling language for importing heterogeneous data representations into a common object-oriented modeling environment. We also introduce a mediation specification language for specifying the structural and semantic differences between the data representations of each pair of heterogeneous systems and their resolution methods. The reusable mediation specifications in a multilevel mediation hierarchy allow a mediated heterogeneous information system to achieve extensibility and scalability. In an implemented prototype system, mediation specifications are used to automatically generate distributed mediators which, in conjunction with subquery processing components, perform distributed query processing and information mediation among a number of component systems over the CORBA communication infrastructure. In this system, efficiency is gained by build-time compilation of mediation specification into distributed mediation code for execution at run-time.

CHAPTER 1
INTRODUCTION

The rapid advancement of the communication network technology in recent years has made it possible to browse and access data stored in different computers via international networks such as the Internet. However, the access of very specific and useful information stored in disparate information systems, such as DBMSs and file systems, is still not well-supported. Consider the case of sharing global medical information on organ donations. It is important and urgent that any hospital in the world should have access to the global information on available organs and donors in order to arrange timely organ transplant operations for patients who need them. This requires not only the interconnection of multiple, distributed, and heterogeneous information systems but also a distributed query processing facility and an information mediation mechanism to access and resolve the discrepancies of data representations in dissimilar systems.

In an integrated heterogeneous information system, system and data heterogeneities among component systems are the major obstacles toward information sharing. System heterogeneity problems result from different hardware and software platforms of component systems and the use of different information or data models and information systems. The differences in hardware/software platforms are usually resolved by using a common communication infrastructure (e.g., TCP/IP communication Socket, RPC, CORBA's ORB, etc.). However, the differences in information and data models are more difficult to deal with. It is not possible to force all users to use the same data or information model for different applications since one data model which is suitable for one application domain may not be suitable for another. Besides, people working in different application areas may have their own preferences on which models to use. The solution to this problem is either to do pair-wise translations of modeling constructs or to use a common or neutral model, to and from which all data models used by the users are translated. The data model heterogeneity problem needs to be resolved before any successful system integration can be achieved.

Data heterogeneity, on the other hand, is due to different ways of modeling data and different semantics and structural representations associated with data values. Problems in this category have been thoroughly investigated and can be generally categorized in two types: schematic heterogeneity and semantic heterogeneity. Schematic heterogeneity is due to different ways of naming and structuring data; whereas, semantic heterogeneity is due to different representations of data values. Because of these problems, queries issued against multiple information sources can not be processed directly and data returned from them can not be readily used. Query and data conversions are needed to bridge the naming, structural and semantic gaps. Two basic approaches have been taken to deal with the problems of data heterogeneity.

The traditional approach is to establish a global schema which reconciles the naming and structural conflicts and unifies the semantic representations of heterogeneous component systems [CHE88]. This integrated global schema is then shared and referenced by the users to pose queries. Thus, all conflicts and differences are resolved at the time of schema design and integration. The major drawback of this approach is that the shared integrated schema forces the users to view data in the same way instead of the ways that are familiar to them.

In contrast to the traditional schema integration approach, more recent thinking is to "mediate" dissimilar data representations instead of "integrating" them. This mediation approach is typically done by using some mediation rules or specifications which are used to resolve various kinds of conflicts among component systems at run-time. A mediated information system allows the users to see data in their own views. They can issue queries based on their own views and receive data in representations that are familiar to them. Furthermore, the mediation approach provides better support for system extensibility and scalability because, unlike the schema integration approach, adding new component systems to the heterogeneous system can be done by changing or adding mediation rules or specifications instead of redesigning or modifying the integrated schema. Due to the above advantages, we have adopted the mediation approach in this research work.

Even though several recent mediation research efforts have reported some interesting results which shall be surveyed in the next chapter, two important problems still need to be explored. First, how do we solve both system and data heterogeneity problems in a unified framework? Second, how do we implement an information modeling and mediation system that is efficient and scalable? Our mediation research focuses on developing effective and efficient solutions to these problems.

Our mediation strategy is outlined as follows. First, to resolve system and data heterogeneities, the specifications of component systems and data are uniformly modeled in an object-oriented, semantically-rich common modeling language, called NCL. Second, to resolve data heterogeneities, we have designed a high-level mediation specification language. It is used to capture the interrelationships among the data specifications of component systems and the information for resolving both schematic and semantic heterogeneities. This language can be applied in a multi-level mediation hierarchy in which the upper-level mediation specification can be incrementally generated by reusing its lower-level mediation specifications. Third, the mediation specification is compiled to generate a set of object classes and some implementations defined in NCL. These generated classes which we shall call "mediation elements" model the distributed query processor, mediators, and subquery processors. They contain active mediation rules which are triggered to perform mediation operations at run-time. Finally, the NCL specifications of the component systems and the mediation elements are combined to form a mediated global schema which is then compiled by the NCL compiler to generate executable program code for mediated distributed query processing.

The term "mediation", first proposed by Wiederhold [WIE92], was defined very broadly [WIE94, WIE95]. The suggested services to be provided by the mediation system may include heterogeneity reconciliations, query optimization, security rule enforcement, negotiation, etc. However, as pointed out in the literature [BRE90, CHAL94, CHAT91, GOH94, HAM93, KIM91, SU93, VEN91], the most critical problem in a heterogeneous information system is the problem of resolving various kinds of data heterogeneities. This problem has not been effectively solved, especially in a large-scale system environment. Therefore, our mediation research focuses on "information mediation" for resolving the data heterogeneity problem, within the context of global query processing.

The remainder of this dissertation is organized as follows. Chapter 2 contains a survey of related research works pertinent to information modeling and mediation system design. Chapter 3 describes the general framework of this mediation system, explaining the build-time system architecture, user's views in issuing queries and the run-time communication architecture. Chapters 4 and 5 detail the common modeling language, NCL, and the mediation specification language, MSL, respectively. Chapter 6 explains the approach taken to implement the mediation system. It covers the translator/compiler developments of the two languages. Chapter 7 describes the run-time mediation rule execution and distributed query processing, and presents some experimentation results using a query example. Chapter 8 summarizes the main contributions of this research and discusses possible directions for future work.

CHAPTER 2
SURVEY OF RELATED WORK

This chapter presents a survey of related work pertinent to this research. Since the focus of this research is on information modeling and mediation techniques, we shall survey the related works in these two areas below.

2.1 Information Modeling

Two independent efforts on standards are particularly relevant to the development of information modeling facility; namely, the efforts of the Object Management Group (OMG) and the International Standard Organization's Committee on the STandard for the Exchange of Product model data (ISO/STEP).

OMG is formed by a consortium of over five hundred industrial companies which aims to define and develop object-oriented technologies for achieving interoperability among dissimilar computing platforms. It developed a Common Object Request Broker Architecture (CORBA) [OMG91] for achieving object interoperability. The interfaces of all objects of interest are specified in a common Interface Definition Language (IDL) [SOM93, OMG91]. An interface specification is compiled to generate program skeletons and stubs for inclusion into the server programs and the client programs, respectively. At run-time, a software or human client can make requests for object services, which are dispatched to the proper servers in a heterogeneous network, thus achieving client/server interoperability.

The ISO/STEP community, on the other hand, emphasizes the development of standards for product modeling and product data exchange. One of its major efforts is the development of an information modeling language named EXPRESS [ISO92]. EXPRESS provides a rich set of constraint specifications by using keywords, functions, procedures and constraint rules. It is a powerful information modeling language. The language has been widely accepted and used by a number of product design and manufacturing communities.

In spite of the individual success and acceptance of these two standards efforts, the result produced by each does not adequately solve the data sharing and program interoperability problems found in a heterogeneous environment. For example, the underlying object model of OMG's IDL is that of C++. While it may be adequate for achieving program interoperability (since the underlying data models of most of the existing object-oriented programming languages have the similar modeling power as IDL), the semantics captured by IDL is not rich enough for modeling complex objects processed by many existing application systems. When IDL is used as the common modeling language to model and encapsulate the objects and object services of an existing application system (e.g., a relational database application or a CAD application), much of the semantics of the objects cannot be captured explicitly because IDL does not have the necessary modeling constructs for capturing constraints by keywords and/or by integrity rules. Thus, much of the needed semantics have to be embedded in the application code. Recognizing the limitations of the object model underlying most of the existing object-oriented DBMSs, the Object Data Management Group (ODMG) [ODM93] and others (e.g., [FLO95]) have made some effort to extend the object model's capabilities to capture some semantic constraints of data such as "key" and "inverse attribute" constraints. However, these extensions are still far from meeting the actual needs for modeling complex objects found in many application domains.

On the other hand, although EXPRESS is semantically much richer than IDL and has an object-oriented flavor, it is not an object-oriented information modeling language because it does not support the encapsulation of behavioral properties of objects. Unlike methods found in an object-oriented model, functions and procedures defined in EXPRESS are global properties in a schema and are used in rules for constraint specifications.

Our information modeling language is designed to combine the behavioral specification of IDL and the information modeling power of EXPRESS into a single well-integrated object model and modeling language. The resulting object model and language can be ideal for modeling objects and object services in a network of heterogeneous computing systems. The language can be designed to conform to the two standard languages (IDL and EXPRESS) semantically and, as much as possible, syntactically. In addition, our modeling language has additional new features including more semantic associations and constraints, and behavioral modeling using Event-Condition-Action-AlternativeAction rules.

2.2 Mediation System

Several research efforts on information mediation have been reported in recent years [AMB94, CHAW94, GOH94, LU95, FLO96, SAU96]. They can be categorized into two types. One type aims to mediate the differences between the integrated global view and the views of component systems. In the systems reported in [AMB94, CHAW94, GOH94, LU95, FLO96], a global integrated schema is used to model the global data resources and the mediation is between the component systems' data representations and the global data representation. The approach requires only 2N mediation mappings where N is the number of component systems. This approach is very similar to the traditional integrated approach except some kind of mediation specification is used to facilitate data translations. The other type of mediation work [QIA93, SAU96] mediates the differences between the views of each pair of component systems. It emphasizes the support to users' preferences in viewing data in their own ways. It has the advantages of extensibility and scalability addressed previously. Our mediation work belongs to the second type.

In terms of mediation knowledge representation, pattern-based logic rules [AMB94, CHAW94, GOH94, LU95, FLO96] have been widely used by different research groups in the A.I. community. However, these logic rules are low-level in their representations and are difficult for the user to understand, modify and maintain. Their executions are based on a repetitive inferencing process which, generally speaking, is not very efficient. Their underlying inference engines are based on a run-time interpretation of rules which makes it difficult to achieve high performance. In our work, we use a high-level mediation specification language for better readability and maintainability. Its implementation is based on a compilation approach to generate event-triggered mediation rules. The mediation rules are, in turn, compiled to generate execution code to carry out the mediation process at run-time. Compared to the interpretation approach, this compilation approach can be much more efficient.

Almost all the above mediation works store and manage their mediation rules or specifications in a centralized system. This strategy is not suitable for managing a large number of rules. When the number of component systems becomes large, adding or modifying rules is an expensive process. A better strategy [CHAW94] is to distributedly maintain the mediation rules in a hierarchical structure. Each mediator in the structure integrates its next-level systems and exports an integrated view to its upper-level mediators. That is, each mediator only performs the mediation between its integrated view and the views of its next-level systems. The mediation rules are thus distributed in different mediators and the complexity of rule management is reduced. However, there are two major shortcomings in this approach. First, the problems of the integrated view are not resolved. Second, due to the point-to-point communication between each mediator and its children, the efficiency of query processing is affected since queries issued against the upper-level mediators may need to be passed via many intermediate mediators to reach the information sources. Our work remedies the two shortcomings by using a different hierarchical mediation strategy. Instead of creating multiple integrated views, we build up a mediated global schema based on an object-oriented hierarchy. Different from [CHAW94], each mediation component in this hierarchy contains mediation classes to connect and mediate related entities of its underlying component systems, without exporting an integrated view. Since there is no view integration performed, user's preferences on viewing data are not affected. In addition, to improve the query processing efficiency, the translation of the mediation specification is done by generating executable code which directly dispatches queries to the information sources. Furthermore, to achieve better performance in distributed query processing, the mediation method code generated by the compiler are distributed in multiple mediators, each of which is linked with the code of an information source. Unlike most of the mediation works which overload a centralized mediator, the mediation tasks are distributed among and processed by component systems that contain the relevant data.

CHAPTER 3
A GENERAL FRAMEWORK OF THE MEDIATION SYSTEM

This chapter presents a general framework of the mediation system. Section 3.1 describes the system architecture from the build-time perspective. Then, based on the architecture, Section 3.2 explains the users' views of issuing global queries. Section 3.3 describes the CORBA-based run-time query processing environment.

3.1 System Architecture

Figure 3.1 depicts the general build-time architecture of our information mediation system. As shown at the lower part of the figure, component systems could be heterogeneous information sources, such as databases modeled using different data models (e.g., relational, hierarchical, network, object-oriented, etc.) or file systems. To resolve the problem of dissimilar data models, an O-O common modeling language is used to uniformly model the data and software resources of these component systems to form a set of component schemas. A wrapper is used to do the mapping between the object-oriented representation and the native representation of each component system.

A high-level information mediation language is then used to specify the differences and similarities of each pair of component schemas and the ways to mediate their differences. In the mediation specifications, *mediation classes* are defined over the related entity classes of different component schemas, and mediation clauses are used to specify their interrelationships and the conversion methods needed to mediate their differences.

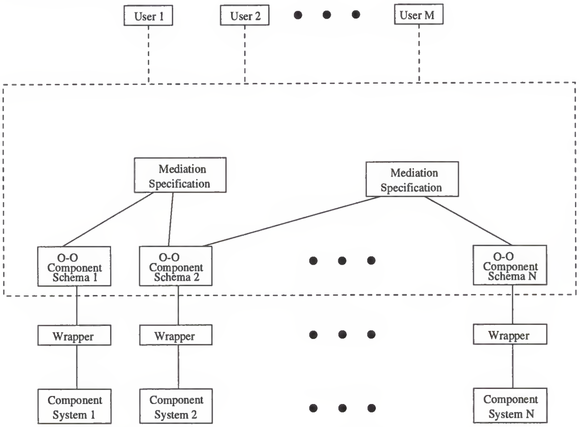


Figure 3.1. System Architecture of Schema Mediation

The component schemas and the additional mediation classes defined in the mediation specifications form a mediated global schema (as shown in the dotted rectangle). This mediated global schema allows users to have their own views of data because each component schema is a part of the mediated global schema. The users of different component systems can issue global queries based on the views that are familiar to them.

3.2 Users' Views for Issuing Queries

In the mediated global schema, there are two ways of issuing global queries to retrieve data from multiple information sources. One way is based on the component system schemas; the other way is based on the mediation classes defined in the mediation specifications.

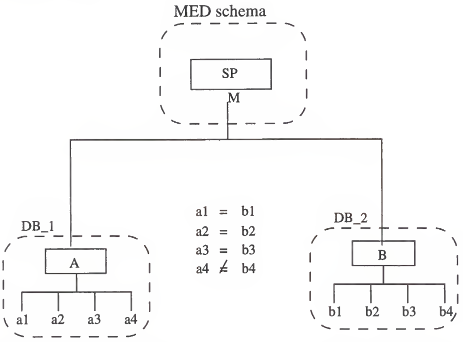


Figure 3.2. Example of Creating a Mediation Class on Top of Two Related Classes

To illustrate these two views for issuing global queries, we use the example schema given in Figure 3.2 which contains a mediation class SP defined on top of two related classes, A and B, in component schemas DB_1 and DB_2, respectively. The M association from the mediation class SP to classes A and B is conceptually similar to the generalization (i.e. G association). The only difference is that the mediation class SP using an M association upward inherits all attributes of its constituent classes (i.e., A and B). In this example, three attributes (a1, a2 and a3) of class A are equivalent to three corresponding attributes (b1, b2 and b3) of class B, even though their attribute names are distinct. Attribute a4 of class A and attribute b4 of class B are distinct properties of these two classes. A user can query the system based on the following two views.

Query Based on Component Schemas

A global query can be issued based on the view of either DB_1 or DB_2. Below is a global query issued based on the DB_1's view. The query is posed in an extended version of the object query language (OQL) reported in [ALA89].

```
CONTEXT c:DB_1::A
WHERE c.a3 >= "100"
RETRIEVE c.a1, c.a3, c.a4
```

This query is to retrieve data relevant to the attributes a1, a3 and a4 of class A with the constraint, A.a3 >= "100". In the query, c is a range variable representing all instances which are accessible from all the component systems based on the view of class A in DB_1. Since the mediation specification specifies that objects in DB_1::A and DB_2::B are related objects belonging to the same mediation class even though they have different attribute names, they should all be retrieved based on the view of DB_1. To achieve this, a subquery is generated to retrieve data of class B in DB_2. To enable the subquery processing in the component system DB_2, a mediator which is generated based on the mediation specification is linked with the legacy system that manages DB_2. The mediator would modify the subquery to meet the naming and structural requirements of DB_2 and to be processed by the legacy system. It also would convert the data retrieved from DB_2 into the data specification of DB_1 before being merged with the data retrieved from DB_1. The merged data form the final query result. Similarly, a global query can be issued based on the DB_2's view to retrieve data that are stored in DB_1.

Query Based on Mediation Extension

Based on the preceding view, the attribute b4 of DB_2 is not visible since it is related to class A. To retrieve this or other distinct attributes of DB_2, the user's view should move up to the mediation class (i.e., SP) which upward inherits all the attributes of its related constituent classes. The following OQL query issued against the mediation class SP would retrieve A.a1, A.a3 and values of distinct attributes (i.e., DB_1::A.a4 and DB_2::B.b4).

```
CONTEXT c:MED::SP
WHERE c\DB_1::A.a3 >= "100"
RETRIEVE c\DB_1::A.a1, c\DB_1::A.a3, c\DB_1::A.a4, c\DB_2::B.b4
```

The backslash symbol used in the query is a group identifier used to indicate from which classes the attributes are to be retrieved. It is to avoid the ambiguity resulting from having the same attribute names in different classes with different meanings (i.e., homonyms). Note that all the upward inherited attributes of the mediation class SP are visible to a query issued against SP. Thus, for those equivalent attributes (e.g., a1 and b1) upward inherited by the mediation class, the user can choose which attribute representations to see the data by specifying the proper attribute names in the RETRIEVE clause of the query. For example, in the RETRIEVE clause of the above query, c\DB_1::A.a1 can be changed to c\DB_2::B.b1 to retrieve the data in b1's representation.

3.3 CORBA-based Client/Server Query Processing

In a heterogeneous information system, component systems are physically distributed and interconnected by a communication network. In our work, the information mediation system uses CORBA's Object Request Broker (ORB) as the communication infrastructure.

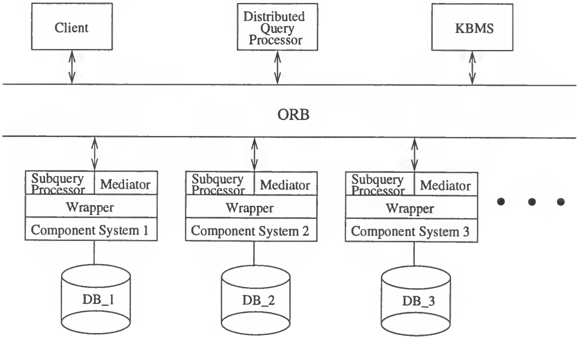


Figure 3.3. CORBA-based Mediation System Architecture

As shown in Figure 3.3, component systems communicate with each other via an ORB. In this communication infrastructure, the user or application program can request services from other component systems by message passing to invoke methods. The method being invoked can exist either locally or remotely. The responsibility of the ORB is to dispatch the messages to the appropriate servers in a transparent manner. The activated method may again invoke other local or remote methods to satisfy the service requirements.

Our information mediation system, as shown in Figure 3.3, consists of Distributed Query Processor (DQP), client, KBMS and information sources (i.e., DB_1, DB_2 and DB_3) whose program codes are linked with generated Subquery Processors (SQPs) and Mediators. At build-time, each of them provides the interfaces to its services which are defined in an NCL component schema. A mediated global schema combining these NCL component schemas is developed and compiled to generate program bindings which include implementation program skeletons and stub header files. The program skeletons are used by the servers into which the program code is inserted; whereas, the stub header files are included in the client programs which request these services. Then, the programs of both server and client sides are compiled and dynamically linked together in the ORB network.

At run-time, the user or application program can issue a global query as a parameter value to the DQP by calling its key method (global_query_execution in our implementation). Once this method is invoked, the processing of the global query is achieved by the collaboration of these component systems (i.e., DQP, KBMS, and SQPs and Mediators), each of which provides part of the distributed query processing and mediation services.

CHAPTER 4
NCL: THE COMMON MODELING LANGUAGE

This chapter presents a common modeling language, NCL, for modeling data and software modules of component systems in a heterogeneous network. The language is developed under a program project funded by DARPA named National Industrial Information Infrastructure Protocols (NIIIP) and the language is thus called NIIIP Common Language (NCL). Section 4.1 first explains the need of a common modeling language. Then, the description of the common language, including its features and syntax, is covered in Section 4.2. Section 4.3 gives an example of using this language to model component systems.

4.1 Need of a Common Modeling Language

In order to resolve the data/information model heterogeneity problems, a widely used approach employs *wrappers*. A wrapper is a program which translates the modeling constructs of one data/information model to those of another. If a heterogeneous information system has N component systems which use N different data/information models, one approach is to write N∗(N-1) wrappers to do pair-wise translations. However, this is not effective. A better and commonly used approach is to introduce a common or neutral model, to and from which N models are translated. This approach requires only 2N wrappers. It is adopted in our work.

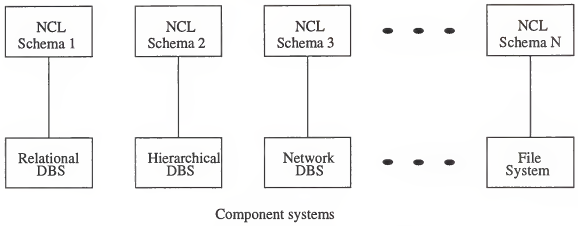


Figure 4.1. Using NCL to Model Heterogeneous Information Resources

To model the information resources of different component systems, the common model needs to be semantically richer than the data/information models used by the component systems to avoid semantic losses in translations. Its modeling language should conform as much as possible to the languages introduced by the standard community. In our work on mediation, we have designed a common modeling language, named NCL [SU96], to model the information resources of component systems, as illustrated in Figure 4.1.

4.2 Description of NCL

NCL is an integration of the language features of CORBA's IDL [SOM93], ISO's EXPRESS [ISO92] and K.3 [SHY96, ARR97]. K.3 is the third version of an implemented knowledge base programming language developed at the University of Florida. NCL's underlying object model is the extensible object model of K.3, which is founded on the concept of objects and object associations introduced in the Object-oriented Semantic Association Model (OSAM* [SU89]) and its algebra and calculus [SU93, KAM94]. NCL uses 1) the method specification facility of IDL, 2) the type and entity specifications and the keyword constraints of EXPRESS, and 3) the knowledge rule specification, language extensibility features, and association type specifications of K.3. The overall structure of NCL is shown below and its complete BNF syntax rules can be found in Appendix A.

```
(* SCHEMA declaration. *)

(* The SCHEMA class has an inclusion relationship with its component class
types *)
DEFINE SCHEMA schema_id;

END_DEFINE;

(* TYPE declaration *)
DEFINE TYPE type_id = underlying_type IN schema_id;
     WHERE  (* domain rule in TYPE *)
     rule_label_1: expression_1;
     ...
     METHODS:
     EXCEPTION exception_id (var_1:type_1;..);
     ...
     METHOD [ONEWAY] method_id
          ([IN|OUT|INOUT] para_id:para_type; ...): return_type
          [RAISES (exception_id, ...)];
     ...
END_DEFINE;

(* ENTITY declaration *)
DEFINE ENTITY entity_id IN schema_id;
     SUPERTYPE OF (supertype_expression)  (* supertype declaration *)
     SUBTYPE OF (subtype list)       (* subtype declaration *)
     attr_id: [OPTIONAL] base_type [WHERE ([TOTAL]
                                 [CARDINALITY([L1:U1]:[L2:U2])]) ];

     ...
     DERIVE
        ...
     INVERSE
        ...
     UNIQUE
```

```
            ...
         [WHERE  (* domain rule in ENTITY *)
         rule_label_1: rule_expression_1;
         ...
         ASSOCIATIONS:   (* Other association types *)
         INTERACTION OF (attr_link_1:Entity_1;attr_link_2:Entity_2;...)
              CARDINALITY
              (attr_link_1:attr_link_2= [L1:U1]: [L2:U2];...);
              ...
         ...
         METHODS:        (* method declaration *)
         EXCEPTION exception_id (var_1:type_1;..);
         ...
         METHOD [ONEWAY] method_id
              ([IN|OUT|INOUT] para_id:para_type; ...): return_type
               [RAISES (exception_id, ...)];
         ...

         (* Local rule declaration *)
         RULES:
         RULE rule_id;
           [TRIGGERED triggered_time trigger_operation, triggered_time
            trigger_operation, .....]
           [CONDITION condition_clause]
           [ACTION
              statement_list]
           [OTHERWISE
              statement_list]
         END_RULE;
         ...
END_DEFINE;

(* Global RULE declaration *)
RULE rule_id;
    [TRIGGERED triggered_time trigger_operation, triggered_time
      trigger_operation,.....]
    [CONDITION condition_clause]
    [ACTION
        statement_list]
    [OTHERWISE
        statement_list]
END_RULE;

(* Method implementation *)
METHOD [class_id::]method_id;
    [LOCAL
      local_var_declaration;
    END_LOCAL;]
    [statement_list]
END_METHOD;
```

The above structure gives the skeletal specifications of schema, type, entity, global rule, and method implementation. The symbols and clauses enclosed in a pair of brackets [ ] can be optional when other mandatory or optional symbols and clauses are present. The syntax of NCL resembles that of EXPRESS; however, minor changes have been made to EXPRESS in order to introduce the language extensibility feature and other additional features of NCL.

NCL provides the constructs for defining schemas, data types and entity types. Their definitions are enclosed by a pair of keywords, DEFINE and END_DEFINE. In NCL, a schema is treated as a first class object. The information resources of different component systems are defined by separate schemas, and their interrelationships are specified by associations among these schema objects, much the same way as the associations among data objects. In NCL, ENTITY, TYPE, SCHEMA, and any other new class types are treated as identifiers (i.e., a parameter of the keyword DEFINE) instead of keywords. The keyword DEFINE signals the compiler that the identifier following it should match with a class type defined in the meta-model. Other class types can be added to the meta-model by the knowledge base customizer who customizes the extensible object model to meet the modeling needs of an application domain. Any identifier can be used as the name of a class type as long as the name used in NCL is consistent with the name used in the meta-model in which the class type is defined. Thus, by adding new class types into the meta-model, we can extend the modeling capability of NCL, thus achieving NCL's class-type extensibility.

The definitions of TYPE and ENTITY are similar to those of EXPRESS except that their semantic contents have been enriched. In the TYPE specification, the behavioral properties of a data type are specified in terms of methods which have the same semantic contents as IDL's interface specifications.

In the definition of an ENTITY, the SUPERTYPE/SUBTYPE specification is the same as EXPRESS. The attribute specification and frequently-used constraints defined by keywords (e.g., UNIQUE, OPTIONAL, DERIVE, and INVERSE, which we shall call "keyword constraints") are the same as EXPRESS. Additional constraints, such as 1) TOTAL (i.e., the total participation constraint which specifies that all the objects in the class from which the attribute draws its values have to associate with some objects defined by the attribute), 2) CARDINALITY (i.e., cardinality mappings between the class in which the attribute is defined and the class from which the attribute draws its values), and 3) other user-defined constraints associated with the attribute, are specified as identifiers which follow the keyword WHERE. Constraints which are applicable to the entire entity class are also defined by identifiers following the keyword WHERE, as shown in the line with the comment (* domain rule in ENTITY *). All the keyword constraints used in the language must match with the constraint types defined in a meta-class ConstraintType of the meta-model. In that meta-class, a parameterized rule(s) is used to define the semantics of a keyword constraint type. For example, the constraint type called TOTAL is defined by a parameterized rule which formally specifies that all the objects of the underlying domain of an attribute have to be the attribute values of some entity instances. When a schema is compiled, the rule(s) is bound to those attributes or object classes that use the constraint type. If a new constraint type and its semantic specification in terms of parameterized rules have been added to the meta-model by the knowledge base customizer, the constraint type can be used in a schema declaration without having to change the NCL compiler.

In an entity class declaration, a number of methods and knowledge rules can be defined which are used for processing the instances of the entity class. The method and exception specifications are borrowed from IDL. However, the syntax has been changed to conform to the syntax of K.3. The rule specification is borrowed from K.3. Different from the constraint rules of EXPRESS, which are used for specifying inter-attribute constraints, rules in NCL are Event-Condition-Action-AlternativeAction rules (or ECAA rules). An ECAA rule contains 1) an event (or Trigger) specification, 2) a condition specification which may involve the verification of a complex pattern of object interconnections in multiple object classes and/or a complex quantified expression that involves multiple attributes of different classes, 3) an action specification which specifies a list of system-defined operations (i.e., retrieval and manipulation operations) and/or user-defined operations (i.e., methods) that should be processed if the condition is evaluated to True, and 4) an alternative action specification whose operations are to be processed if the condition is evaluated to False.

The CAA parts of a rule can be triggered Before or Immediate-after the occurrence of a trigger_operation specified in the event part of the rule, or After a transaction is ready to commit. A triggered operation may in turn trigger other rules. The automatic enforcement of these rules makes the underlying system active. It has been widely recognized that the concepts and techniques of agents [FIN93, GEN94, LAN92], mediators [WIE92, WIE94, WIE95] and negotiators [LAN93, MOE92] are very useful for achieving information and program sharing in a complex, heterogeneous computing environment. The rule specification mechanism of NCL is useful for implementing agents, mediators and negotiators in such an environment because their implementations can make use of the active capability of monitoring events and automatically causing some intelligent behaviors to be carried out. It should also be emphasized that the rule specification language of NCL is far more powerful than the constraint rule of EXPRESS because the latter does not have the event/trigger specification capability nor the specification of complex conditions which involves the processing of the attributes and object instances of multiple classes. Although EXPRESS allows the inclusion of functions and procedures in constraint specifications (e.g., $f(x) = 2$), they are not operations or methods that can be activated based on the result of a condition evaluation.

The ASSOCIATION specification in the ENTITY declaration is borrowed from K.3. EXPRESS has the "Generalization" association in the form of SUPERTYPE/ SUBTYPE construct and the "Aggregation" association in the form of attributes;

however, different enterprises may have the need to model different types of objects and their interrelationships. For example, it is important to model a workflow in terms of control association types such as Sequential, Parallel, Synchronization, Dataflow, etc., that connect the activities or processes in a workflow model. The ASSOCIATION specification allows different association types to be defined in an entity class. In the example schema, INTERACTION is treated by the compiler as an identifier which matches with the definition of this association type in the meta-model. In addition to the Interaction Association, other association types and their semantic properties can be defined by the knowledge base customizer in the meta-model so that they can be used by the users of NCL. Similar to class types and keyword constraint types, all association types are defined in the meta-model in terms of parameterized rules. These rules are converted into bound rules at schema compilation time and incorporated into the object classes that make use of the association types. At runtime, these rules are used to enforce the semantics of the association types. This feature of association-type extensibility allows any semantic relationships between or among object classes (thus, among their instances) that are frequently used in an application to be defined as association types. Once defined, the user can use these association types to relate object classes without having to repeatedly specify their rules in all the classes that involve in these types of associations.

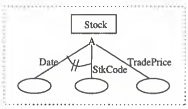
Global rules are those rules which are applicable to all objects. If some rules are only applicable to the objects of some classes, they can be defined in a superclass of these classes and be inherited by them. This is similar to the definition of common attributes and/or methods in the superclass of a number of subclasses.

The method implementation provides the actual program code for implementing the corresponding method specification. It can be coded in any programming language. In our work, we adopted the programming language constructs of K.3 in NCL and used them to implement methods. Therefore, with the definition facilities shown in the above structural declaration and the programming language facilities adopted from K.3, NCL is a full-fledged, high-level, object-oriented programming language.

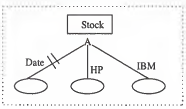
4.3 Example of NCL Schemas

We shall use an example to show how the NCL is used to model component systems. Due to our emphasis on information mediation, the example given below is not intended to reflect the full capabilities of NCL.

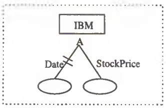
As shown in Figure 4.2, three component systems, DB_1, DB_2 and DB_3, contain related stock information. Each of them contains an ENTITY class related to each other. In DB_1, ENTITY class Stock has three attributes, Date, StkCode and Tradeprice, where Date and StkCode form a composite key. In DB_2, a same-name ENTITY class Stock also has three attributes, Date, HP and IBM, where Date is the key attribute. Attributes HP and IBM contain stock prices of IBM and HP companies, respectively. In DB_3, ENTITY class IBM contains two attributes, Date and StockPrice, where Date is the key attribute. Their NCL schemas are shown as follows:



Component Schema DB_1



Component Schema DB_2



Component Schema DB_3

Figure 4.2. Three Component Schemas Containing Stock Information

```
(* The Specification of component system DB_1 *)
DEFINE SCHEMA DB_1;
END_DEFINE;

DEFINE ENTITY Stock IN DB_1;
  Date: DATE;
  StkCode: STRING;
  TradePrice: STRING;
 UNIQUE
  Date, StkCode;
END_DEFINE;

(* The Specification of component system DB_2 *)
DEFINE SCHEMA DB_2;
END_DEFINE;

DEFINE ENTITY Stock IN DB_2;
  Date: DATE;
  HP: REAL;
  IBM: REAL;
 UNIQUE
  Date;
END_DEFINE;

(* The Specification of component system DB_3 *)
DEFINE SCHEMA DB_3;
END_DEFINE;

DEFINE ENTITY IBM IN DB_3;
  Date: DATE;
  StockPrice: REAL;
 UNIQUE
  Date;
END_DEFINE;
```

The three component schemas represent three different ways that components' data resources are modeled. In schema DB_1, the company name is modeled as an attribute value in the attribute StkCode; however, in schemas DB_2 and DB_3, it is modeled as an attribute and an ENTITY class (i.e., IBM), respectively. Furthermore, the representations of data values in these three component systems are also different. The data representation of the stock price in DB_1 is different from that in DB_2 and DB_3. In DB_1, the stock price is represented in the New York Stock Exchange representation, (e.g., 6\08); whereas, in DB_2 and DB_3, it is in the common decimal representation (e.g., 6.5).

This example, though simple, contains not only naming and structural conflicts but also representational differences. It is general enough to illustrate the mediation process. We shall use this example throughout the rest of the dissertation.

CHAPTER 5
MEDIATION SPECIFICATION LANGUAGE

This chapter presents a Mediation Specification Language (MSL) to capture the mediation information for resolving various data heterogeneity problems and to support system extensibility. Section 5.1 presents a categorization of data heterogeneity (i.e., rationale for the language). Section 5.2 presents the language by describing its design rationale and syntactic structure. Section 5.3 gives an example of using the language to mediate the component schemas. Section 5.4 describes the MSL's property for supporting system extensibility and scalability.

5.1 Categorization of Data Heterogeneity

Data heterogeneity problems have been thoroughly discussed in several publications [BRE90, CHAT91, KIM91, SU91, VEN91, HAM93, CHAL94, GOH94]. Based on the work of Goh, Madnick and Siegel [GOH94], these problems are categorized as schematic heterogeneity and semantic heterogeneity.

Schematic heterogeneity includes two types of problems: naming and structural conflicts. The naming conflicts include the synonym and homonym problems on both attribute and entity type names. The structural conflicts are due to different ways of modeling the same piece of information. For example, in Figure 5.1, the company name, "IBM", can be used as an entity type name, an attribute name, and a value of an attribute in different systems.

Datebase 1 (IBM is an attribute value)

Date	StkName	TradePrice
1/20/95	IBM	50.00
1/20/95	HP	40.00
...	...	...

Database 2 (IBM is an attribute name)

Date	IBM	HP	...
1/20/95	50.00	40.00	...
...	...	...	...

Database 3 (IBM is an entity name)

Entity IBM

Date	TradePrice
1/20/95	50.00
...	...

Entity HP

Date	TradePrice
1/20/95	40.00
...	...

Figure 5.1. Three Examples of Showing Structural Conflicts

Semantic heterogeneity is due to different representations of data values. It includes naming and other representational conflicts. The naming conflicts in attribute values are seen as synonyms (e.g., 'IBM' and 'IBM Corp') and homonyms (e.g., persons with the same name). Other representational conflicts in this category include: (1) measurement conflicts (US Dollar vs. Yen), (2) representation conflicts (New York Stock Exchange representation vs. decimal representation), (3) confounding conflicts (e.g., latest closing price vs. latest trade price), (4) granularity conflicts (e.g., monthly pay vs. yearly pay), and (5) domain type conflicts (e.g., numerical type vs. string type).

5.2 Description of Mediation Specification Language (MSL)

The naming, structural, and representational differences of the component systems as shown in the example of Section 4.3 cause some problems in query processing and data access. Queries processible in one system are not processible in another. Entity and attribute names as well as data values need to be converted to suit the other system. In order to mediate their differences, it is necessary to have a mediation specification to define the interrelationships among the data specifications of component systems and the methods to resolve them. Instead of manually hardcoding the mediation information into a mediation program, the approach taken in this work is to design a high-level mediation specification language to capture that information.

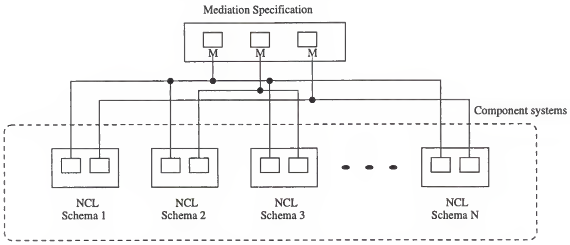


Figure 5.2. Mediation Specification on Top of N Component Schemas

Figure 5.2 gives an overall picture of how the mediation specification is composed and related to the component schemas. In the figure, the specifications of N component systems are modeled as NCL schemas. On top of the component schemas, the mediation specification language is used to specify the interrelationships between related concepts (i.e., ENTITY classes, attributes and values) and the methods that reconcile the differences. The basic idea, also found in [COL91], is to create mediation classes to link related ENTITY classes (i.e., M associations) based on the mediation specification. Attributes of the related ENTITY classes are *upward* inherited by the mediation classes. In the definition of each mediation class, the mappings between related attributes are explicitly specified in the mediation clauses. In addition, the semantic heterogeneities of data values are resolved by specifying the proper conversion methods which convert data values from one representation to another.

It is noted that the mediation class which mediates related ENTITY classes is not intended to create an integrated schema, but rather a *mediated* one. In this mediation system, the mediation class does not unify the names of its upward inherited attributes but contains descriptive clauses to mediate them. Therefore, in this hierarchy, a user's view of data based on each component system is still preserved. A user would issue a query based on his/her preferred view of data (i.e., use the naming, syntactic and semantic representations of the schema of a particular component system). If additional data are stored in other component systems with different representations, the query needs to be modified to conform to those representations before being processed by the component systems. The data returned from them will have to be converted to conform to the user's view. Thus, the mediation process needs to be closely coupled with distributed query processing.

5.2.1 Design of MSL

The design rationale of MSL is based on the following requirements:

- The language must be able to explicitly specify the naming, structural and semantic relationships among heterogeneous systems to capture different types of data heterogeneities as listed in Section 5.1.

- The syntax of the language must be high-level and easy for system designers to specify mediation information.

- The syntactic constructs should conform to the information modeling language (in our case, NCL) so that the translation from the mediation specification to mediation rules and methods of the information modeling language can be done more easily.

5.2.2 MSL Syntax Structure

The overall syntactic structure of the mediation specification language is shown below. Its complete BNF rules are listed in Appendix B.

```
SCHEMA Med_Schema;
 USE FROM schema_1(entity_1,...);
 USE FROM schema_2(entity_2,...);

 ENTITY super_entity_id
  ABSTRACT MEDIATION_TYPE OF (mediation_type_expression);
  ENTITY EQUIVALENCE (sch_1::entity_1,sch_2::entity_2,...);
  ATTRIBUTE EQUIVALENCE [(sch_1::entity_1.attr_1,sch_2::entity_2.attr_2,...);
                                   | (attr_set_1, attr_set_2,...);]
   [VALUE EQUIVALENCE((sch_1::entity_1.attr_1,conv_method(sch_2::entity_2.attr_2),...);
                              | (sch_1::entity_1.attr_1,conv_method(attr_set_2),...);
                              ...
                             ); ]
   [WHERE
        simple_condition;
        ... ]
   ...
 END_ENTITY;
 ...
END_SCHEMA;
```

The mediation specification language conforms to the standard information specification language, EXPRESS (a part of NCL), by using some of its keywords and syntax. Two main constructs of MSL borrowed from EXPRESS are: (1) The USE FROM clause declared inside a schema is to specify the interface between the mediation schema and other component schemas. All the component schemas and their related ENTITY classes are imported by listing their identifiers in the clause. (2) The ABSTRACT MEDIATION_TYPE clause (varied from the ABSTRACT SUPERTYPE clause) in an ENTITY class definition is to define the mediation relation from the mediation class being defined to a set of related ENTITY classes of the component schemas. It also contains a mediation_type_expression that describes the interrelationships between/among these related classes. Relation operators used in the mediation_type_expression include ANDOR, AND and ONEOF which represent the constraints of Set-Intersection, Set-Equality, and Set-Exclusion, respectively. The relationship constraints between/among the classes can contain multiple relation operators. Four additional syntactic constructs of MSL are introduced and explained below.

- The ENTITY EQUIVALENCE clause is used to declare the equivalence relationship between two or more entity types and to resolve the problem of synonymous entity type names. Entity type names enclosed in the clause are declared to be synonyms and are semantically equivalent. The synonym relationships are used to generate code to do query modifications (a method of the Mediator) by the mediation language compiler.

- The ATTRIBUTE EQUIVALENCE clause is used to represent the synonymous relationship among a set of attributes, each of which can be composite. These attributes have the same meaning and their values are inter-convertible. The clause is also used to generate part of the implementation code for query modification. The information of attribute name mappings is embedded in the code of that method.

- The VALUE EQUIVALENCE clause specifies the method of performing data conversions between two different systems. The data conversion method specified in the clause is to convert data of one attribute or a set of attributes into the representation of another attribute. The data conversion is essential to resolve the semantic heterogeneity problem, which may involve both irregular (e.g., synonyms in data values) and regular (e.g., unit or measurement conflict, etc.) data mappings. The synonym problem in data values can be resolved by coding the pairwise mappings of equivalent data values (e.g., the value 'IBM' in system A is equivalent to the value 'IBM Corp' in system B) in the implementation of the conversion method. Similarly, mathematical functions (usually simple ones) can be embedded in the conversion method to deal with regular data mappings. The implementation of conversion methods can be done in NCL or other programming languages.

- The WHERE clause following the ATTRIBUTE EQUIVALENCE clause is to resolve both the homonym problem on values and the structural conflict between two systems. Homonymous values are identified by specifying the equality conditions of key attributes in the WHERE clauses. The attribute values are identical only when the key attribute values are the same. In data modeling, the same piece of information can be modeled in different ways, which causes the problem of structural conflicts. The conversion between two attribute values is possible when a special condition is true (e.g., an entity name is equal to a particular attribute value). It represents a conditional mapping relationship between related classes and is needed in query modification. The WHERE clause is also used to specify the special condition for the mediation.

It is noted that, by default, all entity and attribute names defined in different systems are assumed to be unrelated even though they are the same, unless ENTITY EQUIVALENCE and ATTRIBUTE EQUIVALENCE clauses are used to explicitly specify their synonymous relationships. The homonym problem on attribute values is handled by using the WHERE clause to explicitly specify the equality condition of key attributes as explained above.

5.3 Example of Mediation Specification

The following specification is to mediate the three stock component schemas given earlier:

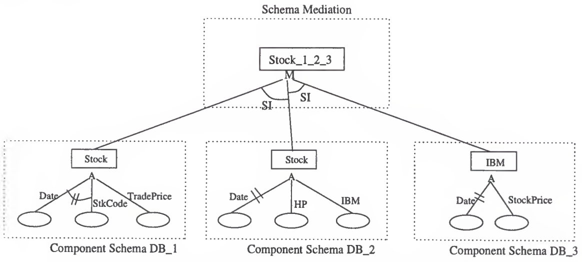


Figure 5.3. Mediation Schema on Top of Three Component Schemas

```
SCHEMA Mediation;
 USE FROM DB_1(Stock);
 USE FROM DB_2(Stock);
 USE FROM DB_3(IBM);

 ENTITY Stock_1_2_3
  ABSTRACT MEDIATION_TYPE OF (DB_1::Stock ANDOR DB_2::Stock
                              ANDOR DB_3::IBM);
  ENTITY EQUIVALENCE (DB_1::Stock,DB_2::Stock,DB_3::IBM);
   ATTRIBUTE EQUIVALENCE (DB_1::Stock.Date,DB_2::Stock.Date,DB_3::IBM.Date);
   ATTRIBUTE EQUIVALENCE (DB_1::Stock.TradePrice,DB_2::Stock.HP);
    VALUE EQUIVALENCE(
     (DB_1::Stock.TradePrice,Decimal_to_NewYork(DB_2::Stock.HP));
     (DB_2::Stock.HP,NewYork_to_Decimal(DB_1::Stock.TradePrice)));
    WHERE
        DB_1::Stock.StkCode = 'HP';
    (* DB_1::Stock.TradePrice and DB_2::Stock.HP are equivalent only when
      DB_1::Stock.StkCode is equal to 'HP' *)
   ATTRIBUTE EQUIVALENCE (DB_1::Stock.TradePrice,DB_2::Stock.IBM,
                          DB_3::IBM.StockPrice);
    VALUE EQUIVALENCE(
     (DB_1::Stock.TradePrice,Decimal_to_NewYork(DB_2::Stock.IBM));
     (DB_2::Stock.IBM,NewYork_to_Decimal(DB_1::Stock.TradePrice)));
     (DB_1::Stock.TradePrice,Decimal_to_NewYork(DB_3::IBM.StockPrice));
     (DB_3::IBM.StockPrice,NewYork_to_Decimal(DB_1::Stock.TradePrice)));
    WHERE
        DB_1::Stock.StkCode = 'IBM';
    (* DB_1::Stock.TradePrice is equivalent to DB_2::Stock.IBM and DB_3::IBM.StockPrice
      only when DB_1::Stock.StkCode is equal to 'IBM' *)
  END_ENTITY;

END_SCHEMA;
```

As shown in Figure 5.3, the three component databases, DB_1, DB_2, and DB_3, contain semantically related stock information. In the above mediation specification, the USE clauses are used to import the three schemas and their related ENTITY classes. The ENTITY class, Stock_1_2_3, is specified as a mediation class of DB_1::Stock, DB_2::Stock and DB_3::IBM with two assumed pair-wise ANDOR constraints of EXPRESS which specify that objects in these pairs of classes can overlap. The attributes of the three related ENTITY classes are upward inherited by Stock_1_2_3. Then, inside the definition of the ENTITY Stock_1_2_3, different mediation clauses, such as ENTITY EQUIVALENCE, ATTRIBUTE EQUIVALENCE, VALUE EQUIVALENCE and WHERE, are used to specify the mediation information needed to reconcile their schematic and semantic conflicts.

For example, the attributes DB_1::Stock.TradePrice and the attribute DB_2::Stock.HP are equivalent when the value of the attribute DB_1::Stock.StkCode is equal to 'HP'. This information is specified in the ATTRIBUTE EQUIVALENCE clause with the WHERE condition, DB_1::Stock.StkCode = 'HP'. The data of the two equivalent attributes are convertible to each other by using two simple conversion methods as specified in the VALUE EQUIVALENCE clause. For example, DB_1::Stock.TradePrice can be converted into DB_2::Stock.HP by using the conversion method, NewYork_to_Decimal.

5.4 Scalable Multilevel Mediation

In a heterogeneous information system, component systems which provide data or services may change over time: old component systems become obsolete and are removed and new component systems that provide more up-to-date information or better services are added. This suggests the need of system extensibility. Furthermore, to achieve a large amount of information exchange and sharing, the number of component systems can become rather large. In this case, scalability becomes a critical requirement. The traditional schema integration approach is not suitable because a potentially large amount of data heterogeneities among different component systems can impede the integration process and become unmanageable.

5.4.1 Multi-level Mediation Hierarchies

The requirements for extensibility and scalability have motivated our work to form a multi-level mediation hierarchy by which both requirements can be met. The mediation specifications that capture lower-level mediation information can be reused to generate the upper-level mediation specifications. The basic idea is similar to the upward inheritance of attributes by a mediation class. The mediation specifications can also be upward "inherited" in such a hierarchy. Since the mediation specifications already capture the mediation information for mediating the component systems in a certain scope, it can be reused to generate the mediation specification of a larger scope of component systems. This reusable property of the mediation specification greatly reduces the effort and cost of composing upper-level mediation specifications.

Figure 5.4 illustrates the idea of reusing the mediation specifications. The mediation specifications defined at lower levels are reused and combined with the top-level mediation specification to generate the new mediation elements to perform the mediation tasks in a broader scope (i.e., containing more component systems). In Figure 5.4, four component systems named DB_1, DB_2, DB_3 and DB_4 are at the level 0 of the mediation hierarchy. Each component system contains two ENTITY classes which are related to the ENTITY classes of other component systems. Then, in level 1, the component systems are mediated by two mediation specifications, 1A

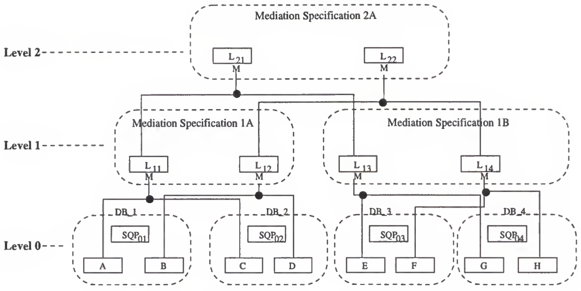


Figure 5.4. Multi-level Mediation: Build-time Hierarchy

(for DB_1 and DB_2) and 1B (for DB_3 and DB_4). Each mediation specification can be compiled to generate the mediation elements that perform the mediation process in its scope. Suppose that the mediation elements for mediating the four component systems is to be created. The two mediation specifications (1A and 1B) can be reused and combined with the new mediation specification, 2A, which mediates the two separate sets of component systems. The advantage of this strategy is that redundant mediation specifications can be avoided. It is noted that, since the compilation approach is taken, the compiler of the mediation language is designed to combine multiple mediation specifications and generate the mediation elements that deal with a broader scope of information mediation.

Figure 5.5 shows the mediation hierarchy after generating three sets of mediation elements 1A, 1B and 2A. At run-time, the DQP of each set of mediation elements works independently to receive and process queries in a different scope. The dotted arrow lines show the method calls invoked directly from the DQP to the SQPs of the

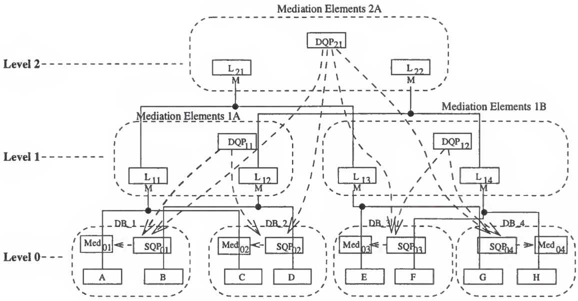


Figure 5.5. Multi-level Mediation: Run-time Hierarchy

component systems (i.e., SQP_{0x}). The method calls are then trapped by mediation rules to invoke local mediators to perform mediation operations.

In this hierarchy, multiple sets of mediation elements are generated to process queries in different scopes of component systems. We believe that this scoped information access has a significant merit because, in most cases, the users may not be interested in retrieving data from all the information sources of the entire mediation system. This multi-level mediation hierarchy can support different levels of grouping component systems and their data resources. In the real world situation, the component systems providing data are usually grouped by certain criteria. For example, information sources in the same geographical area can be grouped together and mediated by a set of mediation elements. Then information sources of neighboring areas can further be grouped to form a larger mediation scope. Queries for retrieving data in a particular scope can then be submitted to the relevant mediation elements for processing.

Furthermore, this multi-level mediation hierarchy can provide better system availability. Compared with a centralized mediation system in which any changes would cause the interruption of mediation services, changes made to the multi-level mediation hierarchy only affect some of its distributed mediation elements. All others can still process queries in their mediation scopes.

5.4.2 Adding or Deleting Component Systems and Mediation Specifications

In the multilevel mediation hierarchy, adding or deleting component systems is done by changing the mediation specifications which have been affected and then recompiling them to generate new mediation elements. At build-time, the mediation hierarchy is a tree structure in which terminal nodes are component systems and non-terminal nodes are mediation specifications. Any addition or deletion of component systems will cause changes to be made in some nodes of the tree.

Generally, there are three alternatives of adding new component systems. To illustrate the three alternatives, we shall use an example. As seen in Figures 5.6, 5.7 and 5.8, a three-level mediation hierarchy (i.e., uncircled part) has existed before appending new component systems. In the lowest level, there are six component schemas and each of them contains an ENTITY class (i.e., L_{01}-L_{06}). The six ENTITY classes are related and mediated by the two mediation specifications, $MSpec_{11}$ and $MSpec_{12}$, with the mediation classes L_{11} and L_{12}, respectively. On top of the two mediation classes, another mediation class L_{21} is defined to mediate the two groups of component systems.

Based on the example, three component systems (each represented as a dotted rectangle) are added and each contains an ENTITY class (i.e.,L_{07}, L_{08} and L_{09}) related to the existing ones. The addition of the new component systems results in three possible types of configurations:

(1) *Appending new component systems to an existing mediation specification*: In this case, new component systems are directly inserted under a node of an existing mediation specification. As shown in Figure 5.6, the three new ENTITY classes are inserted under the node of mediation specification, $MSpec_{12}$. To add the new component systems, the mediation specification $MSpec_{12}$ needs to be changed.

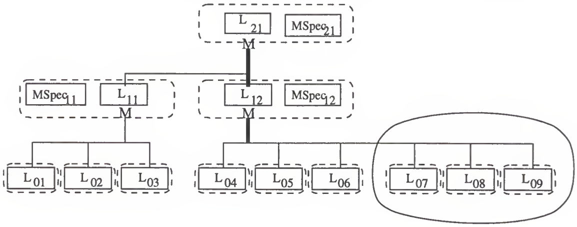


Figure 5.6. Appending New Component Systems to an Existing Mediation Specification

(2) *Creating a new non-root mediation specification*: This alternative creates a new mediation specification to mediate the new component systems and forms a subtree. Then, the subtree is inserted somewhere in the existing mediation hierarchical tree. Figure 5.7 depicts this case of creating a new mediation specification $MSpec_{13}$

to mediate the three new component systems and the subtree with the root L_{13} is inserted under the mediation specification $MSpec_{21}$.

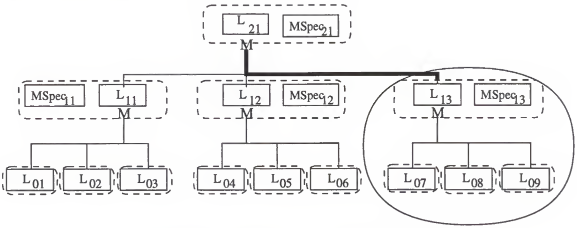


Figure 5.7. Creating a New Non-root Mediation Specification

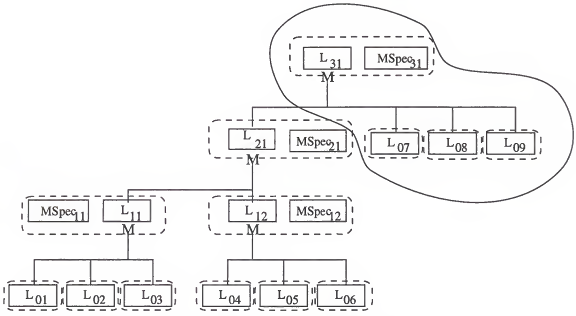


Figure 5.8. Creating a New Root Mediation Specification

(3) *Creating a new root mediation specification*: The last alternative is to create a new mediation specification on top of the existing mediation hierarchical tree and the new component systems. As shown in Figure 5.8, a new mediation specification,

$MSpec_{31}$, is created to mediate the existing mediation tree with the root L_{21} and the three new component systems.

It is noted that the first two alternatives may require some changes on the mediation specifications in the path from the new component systems to the root, as shown by the thick lines in Figures 5.6 and 5.7. Each mediation specification in the path should be modified only if any entity classes of the new component systems relate to the entity classes of its existing component systems; otherwise, no change is needed.

On the other hand, the deletion of component systems is done by modifying the mediation specifications in the path of the mediation hierarchical tree. All mediation clauses containing the entity classes to be removed should be modified properly. Since the mediation specification language is high-level in its representation, it is easier to understand, modify and maintain to meet the change requirements than hard-coded mediator(s).

CHAPTER 6
SYSTEM IMPLEMENTATION

This chapter presents the implementation of the mediation system which uses the two languages described in Chapters 4 and 5. To illustrate the build-time process of the mediation system, the order of presentation is reversed from the order of describing the two languages. The implementation of the MSL translator is described first followed by the implementation of the NCL compiler.

6.1 MSL Translator

The MSL translator translates a mediation specification into a set of mediation elements for supporting run-time mediation and query processing. We shall describe the generation of the mediation elements and give some generated examples below.

6.1.1 Generation of Mediation Elements

The MSL translator takes a mediation specification containing descriptive mediation clauses as its input. The mediation specification is first syntactically and semantically checked, and then produces a parse tree structure of its original representation. By making use of this tree representation, the MSL translator generates a set of mediation elements, as depicted in Figure 6.1. The generated mediation elements are described below.

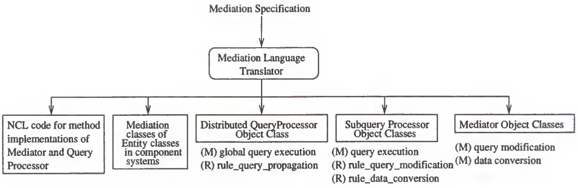

Mediation Specification
Mediation Language Translator
NCL code for method implementations of Mediator and Query Processor
Mediation classes of Entity classes in component systems
Distributed QueryProcessor Object Class
(M) global query execution
(R) rule_query_propagation
Subquery Processor Object Classes
(M) query execution
(R) rule_query_modification
(R) rule_data_conversion
Mediator Object Classes
(M) query modification
(M) data conversion

Figure 6.1. Compilation of Mediation Specification

- **Mediation classes**: These mediation classes mediates the ENTITY classes of different component systems if they contain semantically related objects. The specifications of these mediation classes also capture the set membership constraints among the related ENTITY classes, such as Set-Equality, Set-Exclusion, or Set-Intersection. This information is useful for query optimization (e.g., if two component systems contain identical objects and their data are the same, then there is no need to send a query to both systems).

- **Mediator object classes**: These classes model the mediators which are generated based on the mediation specifications associated with pairs of related component schemas. For the sake of query processing efficiency, the Mediator classes are distributed and linked with different component systems at different sites so that mediation operations can be processed locally. Each Mediator contains the methods that actually perform query modification and data conversion. These methods are invoked by the Subquery Processor of the component system when mediation is needed.

- **Distributed Query Processor (DQP) object class**: The Distributed Query Processor class contains a method which performs distributed query processing in the heterogeneous system. This method with a query as its parameter is invoked by the user/application program. The implementation of this method is based on a built-in query processing algorithm. The DQP class has an ECAA mediation rule which, when triggered, calls a knowledgebase management system (OSAM*.KBMS [SU95]) to obtain the meta data for locating the information sources and propagating subqueries.

- **Subquery Processor (SQP) object classes**: These classes are Subquery Processors generated for the component systems. Each SQP receives a subquery from the DQP and calls the Mediator, if a subquery conversion is needed, to generate a mediated subquery. The SQP then sends the mediated subquery to the wrapper which converts it into a native query, command or API processable by the component system. The SQP object class contains a main method which triggers two mediation rules. One is to modify the subquery to conform to the naming and structural convention of the component system before the main method is executed. The other is to convert the returned data from its local representation to the one expected by the user after the main method finishes its execution.

- **Mediation methods**: Based on the mediation specification (i.e., ENTITY EQUIVALENCE, ATTRIBUTE EQUIVALENCE, VALUE EQUIVALENCE and WHERE), the implementation code for the mediation methods, including query modification, data conversion, etc., can either be generated automatically or provided by the system administrator.

All the above mediation elements including DQP, SQPs, distributed Mediators, mediation rules, and method implementations are generated and uniformly represented in NCL. The NCL code is then translated into executable code (C/C++) by an NCL compiler. The next section will give some examples describing the generations of the Mediator object classes and the mediation rules associated with DQP and SQPs of component systems.

6.1.2 Examples of Generated Mediation Elements

Generation of Mediators

Each mediator, linked to a component system, provides pairwise conflict resolution between the component system and other component systems. The following mediator class of the component system DB_1 is a part of the output generated from the mediation specification given in Chapter 5.

```
DEFINE ENTITY Mediator IN DB_1;
METHODS:
 METHOD modify_query(INOUT ps:Query):VOID;
 METHOD data_conversion (INOUT result:SET OF Data):VOID;
 METHOD convert(INOUT data:Data):VOID;
 METHOD change_name_add_cond(IN s_name:STRING;IN t_name:STRING;
                INOUT ps:Query; IN condition:STRING):VOID;
 METHOD change_name_rm_cond(IN s_name:STRING;IN t_name:STRING;
                INOUT ps:Query; IN condition:STRING):VOID;
 METHOD NewYork_to_Decimal(INOUT data:Data):VOID;
END_DEFINE;

METHOD Mediator::modify_query(INOUT ps:Query):VOID;
  IF (ps.s_schema= 'DB_2' AND ps.t_schema= 'DB_1')
  THEN
    change_name_add_cond ('IBM','TradePrice',ps,'StkCode="IBM"');
    change_name_add_cond ('HP','TradePrice',ps,'StkCode="HP"');
  END_IF;

  IF (ps.s_schema= 'DB_3' AND ps.t_schema= 'DB_1')
  THEN
    change_name_add_cond ('StockPrice','TradePrice',ps,'StkCode="IBM"');
    change_name_rm_cond ('IBM','Stock',ps,'');
  END_IF;
END_METHOD;

METHOD Mediator::data_conversion(INOUT result:SET OF Data):VOID;
LOCAL
  num_column, i: INTEGER;
END_LOCAL;
  num_column := SIZEOF(result);
  FOR i=1 UNTIL i > num_column BY i= i+1 DO
    IF (result[i].s_attr <> result[i].t_attr)
    THEN
      convert (result[i]);
    END_IF;
  END_FOR;
END_METHOD;

METHOD Mediator::convert(INOUT data:Data):VOID;
LOCAL
  i: INTEGER;
END_LOCAL;
  FOR i=1 UNTIL i > data.size BY i= i+1 DO
    IF (data.s_attr = 'DB_1.Stock.TradePrice' AND
        data.t_attr = 'DB_2.Stock.HP')
    THEN
      NewYork_to_Decimal(data.value_set[i]);
    END_IF;

    IF (data.s_attr = 'DB_1.Stock.TradePrice' AND
        data.t_attr = 'DB_2.Stock.IBM')
    THEN
      NewYork_to_Decimal(data.value_set[i]);
    END_IF;

    IF (data.s_attr = 'DB_1.Stock.TradePrice' AND
        data.t_attr = 'DB_3.IBM.StockPrice')
    THEN
      NewYork_to_Decimal(data.value_set[i]);
    END_IF;
    ...

  END_FOR;
END_METHOD;


(* Common data structures of Query and returned data *)
(* object class 'Query' for storing query statement and data result *)
DEFINE ENTITY Query;
    query_node: Query_node;             (* query node *)
    s_schema: STRING;                   (* source schema name *)
    t_schema: STRING;                   (* target schema name *)
    result: SET OF Data;                (* query result *)
END_DEFINE;

(* Object class 'Data' is defined for standard data interchange *)
DEFINE ENTITY Data;
(* The first four are attribute-value information returned by the component
   system.  The last one is stored by the mediator to specify the target
   attribute *)
   value_set: SET OF Generic_Data_Type; (* result values *)
   size: INTEGER;                       (* size of SET *)
   data_type: STRING;                   (* data type *)
   s_attr: STRING;                      (* attribute name in source schema *)
   t_attr: STRING;                      (* attribute name in target schema *)
END_DEFINE;

(* Generic Data Type: a union of all possible data types *)
DEFINE TYPE Generic_Data_Type=
            SELECT OF (string_type, number_type, boolean_type, Data);
END_DEFINE;
```

The above generated class and method definitions make reference to two data structures for recording the query and data. They are modeled by two entity classes, Query and Data. A TYPE class, Generic_Data_Type is also used. The Query class records the tree structural representation of the query (in the attribute query_node), its original schema/view (in the attribute s_schema) and the schema identifier (in the attribute t_schema) of the target system that will process the query. The data received after query processing is stored as a set of Data objects (in the attribute result). The Data class is to record a set of returned values of a particular attribute of the source schema (i.e., a column of returned data). It also contains the attribute name of the target schema. If the source and target attributes are different, their query modification and data conversion need to be carried out. The TYPE Generic_Data_Type is a union structure of all the data types of the returned values.

The object class, Mediator, contains several methods including modify_query(), data_conversion() and convert(), etc. The first two methods are the main methods to perform query modification and data conversion, respectively. The method, convert(), is a subroutine of data_conversion().

As seen in the method implementation code of query_modification(), the query is modified by first matching its source and target schema identifiers to decide which set of query modification operations is to be taken. This query modification method is generated from the specifications of the ENTITY EQUIVALENCE, ATTRIBUTE EQUIVALENCE and WHERE clauses. Since the query is passed as a query tree, the query modification is done by operations on the nodes of the tree (i.e., deletion, insertion, substitution, etc.). Due to possible structural conflicts between component schemas, query modification may not be done by direct name substitutions in tree nodes. The method change_name_add_cond() (its implementation is not shown) is called for the purpose of either conditional or unconditional name changes. If the condition parameter (i.e, the fourth parameter) is passed with a value, the query modifier not only changes names but also adds the condition expression to the query tree. For example, if the source schema identifier is 'DB_2' and the target schema identifier is 'DB_1', the attribute name 'IBM' in the query is replaced by 'TradePrice' and the condition expression 'StkCode="IBM"', is added. Another method change_name_rm_cond() (the implementation is not shown) modifies the query by changing names and also removing the condition expression from the query.

For data conversion, the method convert() is called when a data conversion is required. It compares the source and target attribute names in the Data object with other pairs of attribute names. The generation of the data conversion method is based on the specifications of the ATTRIBUTE EQUIVALENCE and VALUE EQUIVALENCE clauses. If a pair matches the names, a data conversion method is invoked to perform the data conversion on each of the data values in the column. For example, if the source attribute name is 'DB_1.Stock.TradePrice' and the target attribute name is 'DB_2.Stock.HP', the method NewYork_to_Decimal() is invoked to convert the data. The method has to be coded by a program. It can not be generated automatically.

<u>Generation of Distributed and Subquery Processors and Mediation Rules</u>

The distributed query processor (DQP) is a component system which receives and processes global queries against some or all component systems. Based on the mediation specification, the generated class definitions of the DQP and the subquery processor SQP are shown below:

```
(* Definition of object class 'Distributed Query_Processor' *)
DEFINE ENTITY DQP IN DQP_SCHEMA;
     global_query: Query;              (* declaration for global query *)
     subqueries: SET OF Query;         (* declaration for subqueries *)
     prop_subqs: SET OF Query;         (* for propagated subqueries *)
     prop_index: INDEX;                 (* index of propagated subqueries *)
  METHODS:
    (* global_query_execution() is the entry point *)
    METHOD global_query_execution(IN query_txt:STRING) :SET OF Data;

    METHOD query_initialization(IN query_txt:STRING): VOID;
    METHOD syntax_semantic_checking(): BOOLEAN;
    METHOD error_handler(): VOID;
    METHOD subquery_generation(IN query_txt:STRING): VOID;
    METHOD decompose_query(IN query_txt:STRING): SET OF Query;
    METHOD dispatch_subquery(): VOID;
    METHOD assemble_mediated_results(INOUT q: Query; IN subqs: SET OF Query)
                                                : VOID;
    METHOD result_merge_join(): VOID;
    METHOD query_execution(): VOID;
  RULES:
    RULE rule_query_propagation;
    TRIGGER IMMEDIATE AFTER subq_initialization(s)
    ACTION
      prop_subqs:= KBMS::propagate_queries(s);
    END_RULE;
END_DEFINE;

METHOD DQP::global_query_execution (IN query_txt: STRING):
                                   SET OF Data;
LOCAL
  s: Query;
  ps: Query;
END_LOCAL;
  query_initialization(query_txt);
  IF (syntax_semantic_checking() == FALSE)
  THEN
    error_handler();
  END_IF;
  subquery_generation(global_query.query_stmt);

  FOREACH (s:subqueries)
    subq_initialization(s);
    FOREACH (ps:prop_subqs)
      dispatch_subquery(ps);
    END_FOREACH;
    assemble_mediated_results(s,prop_subqs);
  END_FOREACH;

  result_merge_join();
  RETURN (global_query.result);
END_METHOD;
```

```
(* Component Schema of System DB_1 *)
DEFINE SCHEMA DB_1;
END_DEFINE;

DEFINE ENTITY SQP IN DB_1;
  m: Mediator;
METHODS:
  METHOD query_execution(INOUT q_obj: Query_obj): VOID;
RULES:
  RULE rule_query_modification;
  TRIGGER BEFORE query_execution(q_obj)
  CONDITION q_obj.s_schema <> q_obj.t_schema
  ACTION
    m.modify_query(q_obj);
  END_RULE;

  RULE rule_data_conversion;
  TRIGGER IMMEDIATE AFTER query_execution(q_obj)
  ACTION
    m.data_conversion(q_obj.result);
  END_RULE;
END_DEFINE;
```

The DQP class models the distributed query processor. Its main method is global_query_execution(). By calling this method, a global query is submitted, syntactically checked and decomposed into subqueries if the original query requires the processing of data residing in multiple component systems. Each subquery is dispatched to the SQP of a component system by invoking the main method, query_execution. After subqueries finish their executions, returned data are assembled, merged and joined to produce the final result of the original query.

To perform run-time mediation, three ECAA mediation rules are generated to mediate the distributed query processing. They are rule_query_propagation, rule_query_modification and rule_data_conversion. The first rule, rule_query_propagation, is triggered by the DQP immediately after each subquery is initialized and ready for dispatching to the query server of a component system. The action part of the rule contains a call to a method of the KBMS, propagate_queries(), to get the mediation information which relates the data referenced by the subquery to the data stored in other component systems. The subquery and the schema identifiers of these component systems are entered into the data structure defined in the entity class Query. The purpose is to replicate the subquery into as many subqueries as there are component systems that contain related data. However, at this stage, the replicated subqueries may not be ready for processing because the names and structures of the subqueries may not be recognized by the component systems that will process them. The second rule, rule_query_modification, associated with the SQP of a component system, is triggered before each subquery is executed. It checks to see if the source and target schemas/views are different. If they are different, the method query_modification of the Mediator is called to convert the subquery into the representation suitable for the target system. Additionally, for data conversion purposes, this method also inserts the attribute names used in both the source and target schemas for each data object to be returned by the subquery. After the subquery is processed, the data retrieved by the subquery are returned and the after-rule, rule_data_conversion (also associated with the SQP of the component system), is triggered. This rule enforces the action to check each column of data to see whether the data conversion is needed by comparing the attribute names in the source and target schemas/views. If a naming difference exists, a data conversion is performed.

6.2 NCL Compiler

The above generated mediation elements (i.e., mediation classes, DQP, SQPs and Mediators, etc.) and a collection of NCL component schemas combine to form a mediated global schema. The mediated schema is further compiled by the NCL compiler to generate program code which is distributed and linked in the CORBA-based client/server communication environment. Figure 6.2 depicts the general build-time procedure of compiling the mediated global schema into C/C++ program bindings.

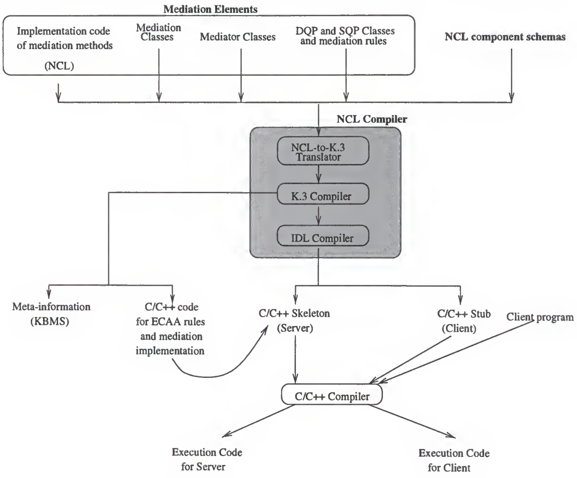


Figure 6.2. Compilation of a Mediated Global Schema

As shown in Figure 6.2, the NCL compiler consists of three parts. First, by using the OSAM*.KBMS as the underlying supporting system, an NCL-to-K.3 translator translates the NCL schema into the corresponding K.3 specification. K.3 is the third version of a knowledge base programming language developed at the Database Systems Research and Development Center [ARR97]. Second, the generated K.3 specification is then compiled by the K.3 compiler to generate IDL specifications and C/C++ program code. Third, an IDL compiler takes the IDL specifications as its input to generate enhanced CORBA program bindings to set up the build-time network environment for both client and server sides. We shall describe the first part of implementations in Section 6.2.1 and the remaining part in Section 6.2.2.

6.2.1 NCL-to-K.3 Translation

Generally, the NCL-to-K.3 translation is a language mapping process. Its implementation is based on a syntactic and semantic analysis of each NCL construct to an equivalent K.3 representation. As listed in Figure 6.3, semantic properties of NCL classes, including attributes, supertype/subtype specifications, additional associations (e.g., INTERACTION), keyword constraints, user-defined rules, and method specifications and implementations, are translated into corresponding K.3 constructs. Some constraints of NCL, such as the UNIQUE keyword constraint, are supported by extending the rule binder of the OSAM*.KBMS meta model. As shown in Figure 6.4, the meta model of the KBMS is based on a self-describing O-O modeling mechanism (i.e., using the model to model itself). Extensions to the meta model can be made by changing the pre-defined meta specifications. Due to the extensibilities of the underlying supporting system, OSAM*.KBMS, additional constructs can be easily added to NCL without entailing changes to the NCL compiler.

NCL Construct	K.3 Specification
SCHEMA	Schema class
ENTITY	Entity class
SUPERTYPE expression	Generalization association
Relation Operators:	Extended Macros:
AND	G_AND
ANDOR	G_ANDOR
ONEOF	G_ONEOF
SUBTYPE expression	Specialization Association
Explicit Attribute	Aggregation Association
OPTIONAL	Default in K.3
TOTAL	Extended Total constraint
CARDINALITY	Extended Acardinality constraint
DERIVE Attribute	Extended parameterized rule Derive
INVERSE Attribute	Extended parameterized rules Inverse1_1,Inverse1_M,InverseM_M
UNIQUE	Extended parameterized rules Unique and CompKey
WHERE clause (Domain rule)	ECAA rule
ASSOCIATIONS:	
INTERACTION	Interaction Association
METHOD declaration	Method signature
EXCEPTION declaration	Entity definition of Exception message
TYPE	Domain class
SELECT clause	Generalization Association
ENUMERATION	Extended macro Enum
METHOD Implementation	Method implementation
local/global ECAA rule	ECAA rule

Figure 6.3. Mappings from NCL to K.3

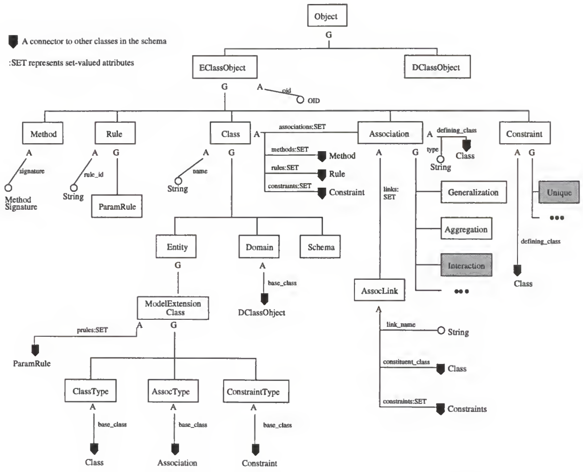


Figure 6.4. Meta Model of the Underlying System, OSAM*.KBMS

In addition to serving as the underlying system for the NCL compilation, the KBMS also serves as a storage manager of the mediated global schema. While compiling a K.3 specification which is translated from an NCL schema, the schema information is also populated in the KBMS. At run-time, the global schema information can be inquired by issuing meta queries against the KBMS. For more details about the KBMS, readers may refer to [SHY96, SU95].

6.2.2 Generation of Enhanced CORBA Bindings

After an NCL schema is translated into a K.3 specification, a K.3 compiler is used to translate the K.3 specification into C or C++ code which contains expanded method calls for implementing the event monitoring and rule processing functions. Figure 6.5 illustrates this compilation process of generating program code and bindings in a CORBA environment. In addition to the C/C++ code generation, IDL specifications for the attributes and the NCL methods are also generated. The reason for generating the IDL specifications is that these methods which provide services for clients may be physically distributed in different servers. The IDL specifications provide the means for achieving interoperability through the ORB.

The final step is to generate the C or C++ programming language bindings for the NCL methods and insert the generated C or C++ code into the skeletons. The bindings are generated by the IDL compiler when the IDL specifications are compiled. Each server site has the bindings and the implementations of its services in its native programming language. A client wanting to use a service will access that service by using the corresponding binding in the programming language of the

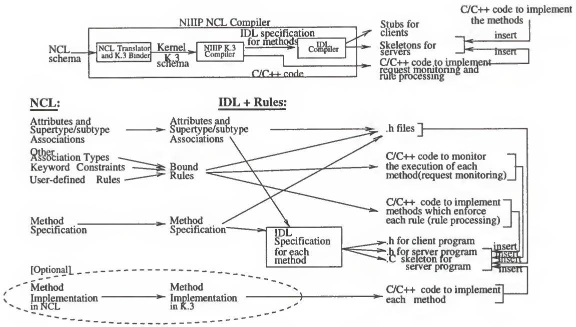

NIIIP NCL Compiler
NCL schema
NCL Translator and K.3 Binder
Kernel K.3 schema
NIIIP K.3 Compiler
IDL specification for methods
IDL Compiler
C/C++ code
Stubs for clients
Skeletons for servers
C/C++ code to implement request monitoring and rule processing
C/C++ code to implement the methods
insert
insert
NCL:
IDL + Rules:
Attributes and Supertype/subtype Associations
Other Association Types
Keyword Constraints
User-defined Rules
Bound Rules
Method Specification
IDL Specification for each method
.h files
C/C++ code to monitor the execution of each method(request monitoring)
C/C++ code to implement methods which enforce each rule (rule processing)
.h for client program
.h for server program
.C skeleton for server program
[Optional]
Method Implementation in NCL
Method Implementation in K.3
C/C++ code to implement each method
insert

Figure 6.5. NCL Schema to C/C++ Code

client. The interoperability is provided by the ORB. Note that the bindings reflect the interface of the original NCL method; however, the implementation of the method consists of the C or C++ code to implement the function of that method, plus the C or C++ code to enforce the before-, after-, and immediately-after rules associated with that method. At run-time, activation of a method will automatically trigger the execution of the methods which implement the CAA parts of rules, which may in turn trigger the method implementation of other rules. It is this enhancement (i.e., the automatic processing of distributed event monitoring and rule processing functions) which makes the ORB "**active**" and provides the **rule-based interoperability** within the CORBA environment (see [SU96] for more details).

Figure 6.6 illustrates the compilation of a method (query_execution) with its associated before- and immediate-after rules (rule_query_modification and rule_data_conversion) into the compiled code for distributed event monitoring and rule processing.

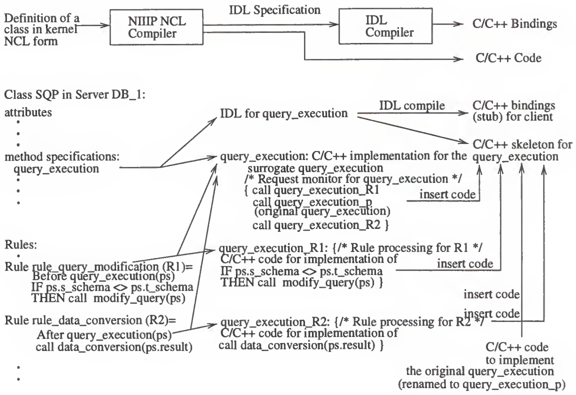

Definition of a class in kernel NCL form
NIIIP NCL Compiler
IDL Specification
IDL Compiler
C/C++ Bindings
C/C++ Code
Class SQP in Server DB_1:
attributes
method specifications:
query_execution
Rules:
Rule rule_query_modification (R1)= Before query_execution(ps) IF ps.s_schema <> ps.t_schema THEN call modify_query(ps)
Rule rule_data_conversion (R2)= After query_execution(ps) call data_conversion(ps.result)
IDL for query_execution
IDL compile
C/C++ bindings (stub) for client
C/C++ skeleton for query_execution
query_execution: C/C++ implementation for the surrogate query_execution /* Request monitor for query_execution */ { call query_execution_R1 call query_execution_p (original query_execution) call query_execution_R2 }
insert code
query_execution_R1: {/* Rule processing for R1 */ C/C++ code for implementation of IF ps.s_schema <> ps.t_schema THEN call modify_query(ps) }
insert code
insert code
query_execution_R2: {/* Rule processing for R2 */ C/C++ code for implementation of call data_conversion(ps.result) }
insert code
C/C++ code to implement the original query_execution (renamed to query_execution_p)

Figure 6.6. Compilation of an NCL Method query_execution() and Its Associated Rules

As shown earlier, the NCL definition of the SQP class of the component system includes the specification of one main method (query_execution), and two mediation rules (rule_query_modification and rule_data_conversion). As described in Chapter 4, NCL rules are Event-Condition-Action-AlternativeAction (ECAA) rules in which the triggering event can be the execution of any method. To illustrate the process of generating the enhanced program binding, we shall use the two mediation rules. They are the before- and after-rules associated with the same method, query_execution, shown below:

```
RULE SQP::rule_query_modification;
TRIGGER BEFORE query_execution(ps)
CONDITION ps.s_schema <> ps.t_schema
ACTION
  modify_query(ps);
END_RULE;

RULE SQP::rule_data_conversion;
TRIGGER IMMEDIATE AFTER query_execution(ps)
ACTION
  data_conversion(ps.result);
END_RULE;
```

Rule_query_modification specifies that before method query_execution is executed, the condition, ps.s_schema <> ps.t_schema, should be checked to see whether the source and target schemas of the subquery are different. If the condition evaluates to True, then the mediation method modify_query is called to perform query modification. Otherwise, no action is needed. Also, rule_data_conversion specifies that immediately after method query_execution is executed, the other mediation method, data_conversion, is called to perform data conversion.

During the compilation of the SQP class by the NCL compiler, a C or C++ method is generated for each rule. For rule_query_modification, the C or C++ code in method query_execution_R1 will check the condition ps.s_schema <> ps.t_schema and call method modify_query if the condition evaluates to True. Similarly, for rule_data_conversion, a C or C++ method query_execution_R2 is generated.

For each method in the class, an equivalent IDL specification is generated. For example, an IDL specification is generated for query_execution. Furthermore, a new implementation of query_execution (i.e., a surrogate query_execution in C or C++ code) is generated. The new implementation consists of three method calls. First, a call to method query_execution_R1 is made to process rule_query_modification (i.e., a before-rule for query_execution). Then, a call is made to the original implementation of the method query_execution, which is renamed as query_execution_p, to perform the query dispatching. Finally, a call is made to method query_execution_R2 to process rule_data_conversion (i.e., an immediate-after rule for query_execution).

In the final step of the compilation process, the IDL compiler is used to generate the C or C++ bindings for all the methods which have been specified in IDL, including method query_execution. After the bindings have been generated, the corresponding C or C++ implementation code for the surrogate query_execution, query_execution_R1, query_execution_p (the original query_execution) and query_execution_R2 can be inserted into the skeleton of query_execution.

CHAPTER 7
EXPERIMENTATION RESULTS

This chapter presents the results of a run-time mediation execution using the implemented mediation system. Section 7.1 demonstrates how the compiled mediation rules are executed at run-time to perform the mediation process. Then, before showing an actual example, a general execution model of distributed query processing and mediation is explained in Section 7.2. Section 7.3 presents an example of a mediated distributed query processing by showing its data and execution flows.

7.1 Run-time Mediation Rule Execution

The enhanced binding example in the previous chapter shows the build-time compilation of the mediation rules and the SQP's method associated with them. This section describes the actual run-time execution of this example. Figure 7.1 illustrates the execution flow of how a service request for query_execution in a component system is monitored to trigger the processing of the mediation rules. First of all, a client makes a query request by using the programming language binding (i.e., an IDL stub) generated by the IDL compiler for the method global_query_execution of the DQP. When the request is made, the ORB would dispatch that request to the DQP server to invoke that method.

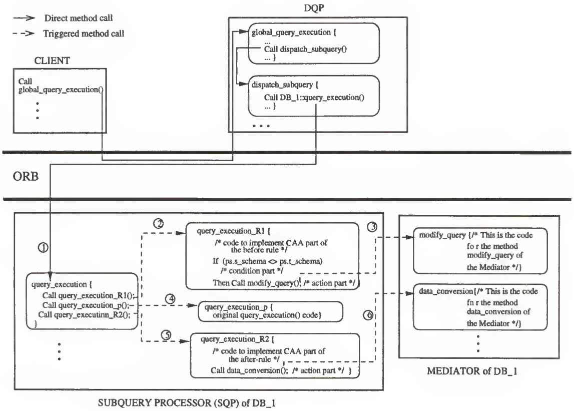

Direct method call
Triggered method call
CLIENT
Call global_query_execution()
DQP
global_query_execution { ... Call dispatch_subquery() ... }
dispatch_subquery { Call DB_1::query_execution() ... }
ORB
①
②
③
④
⑤
⑥
query_execution { Call query_execution_R1(); Call query_execution_p(); Call query_executinn_R2(); }
query_execution_R1 { /* code to implement CAA part of the before rule */ If (ps.s_schema <> ps.t_schema) /* condition part */ Then Call modify_query(); /* action part */ }
query_execution_p { original query_execution() code}
query_execution_R2 { /* code to implement CAA part of the after-rule */ Call data_conversion(); /* action part */ }
modify_query {/* This is the code fo r the method modify_query of the Mediator */}
data_conversion{/* This is the code fn r the method data_conversion of the Mediator */}
SUBQUERY PROCESSOR (SQP) of DB_1
MEDIATOR of DB_1

Figure 7.1. Run-time Execution Flow of Mediation Rules

During the execution of the method global_query_execution, a local call to dispatch_subquery is made to dispatch subqueries to component systems for local query processing. In Figure 7.1, a subquery is dispatched to the DB_1 by calling the method, query_execution (Step (1)); however, in this case the original code for query_execution is not invoked directly. Instead, the generated implementation (i.e., the surrogate query_execution generated by the NCL compiler) is executed. First, the method query_execution_R1 is invoked (Step (2)). The execution of query_execution_R1 involves the checking of the condition for comparing the source and target schemas. If the condition evaluates to True, the action part of the rule is executed and a call to modify_query() is made locally (Step (3)).

After query_execution_R1 is executed to process the before-rule rule_query_modification, a call to the original query_execution (i.e., query_execution_p) is made to execute the code which implements the actual local query processing requested by the client (Step (4)). After the original query_execution has been executed, a call to query_execution_R2 is made to process the immediate-after-rule for performing data conversion (Step (5)).

7.2 Distributed Query Processing and Mediation

This section describes a general model of the global query execution in the system from the data flow perspective. The execution of a global query includes the following steps:

1. *Query decomposition*: Generally, a global query may be complex and involve more than one ENTITY class stored in different component systems. It should

be decomposed before execution. As shown in Figure 7.2, a global query (QueryA) involving two classes is decomposed into two simple subqueries (i.e., each containing a single class), SubqueryA1 and SubqueryA2.

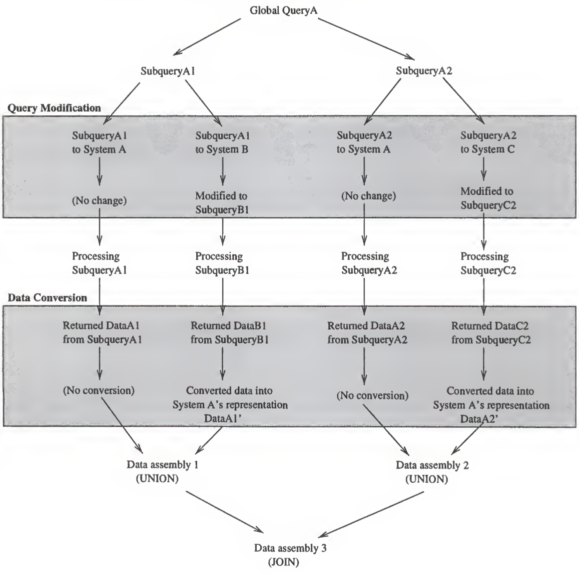


Figure 7.2. Mediated Distributed Query Processing

2. *Subquery propagation*: For each decomposed subquery, the KBMS is inquired to look up the meta information to see whether other information sources contain relevant data to answer the subquery. This meta information is derived

from mediation specifications. In this example, System B also contains data to answer SubqueryA1 while System C also contains data to answer SubqueryA2. Therefore, two replicated copies of SubqueryA1 and SubqueryA2 are generated for System B and System C, respectively.

3. *Subquery modification*: SubqueryA1 dispatched to System B may not be processible since the naming and structural differences may exist between Systems A and B. SubqueryA1 is then modified into SubqueryB1 which is suitable for processing in System B. Similarly, SubqueryA2 is modified into SubqueryC2 for processing in System C. Two other subqueries (SubqueryA1 and SubqueryA2), dispatched to System A, are not changed.

4. *Subquery execution*: All four subqueries are executed locally in Systems A, B and C. Data results of the four subqueries are generated, shown as DataA1, DataB1, DataA2 and DataC2.

5. *Data conversion*: The data retrieved from Systems B and C do not conform to System A's representation. Data conversions are performed to convert DataB1 into DataA1' and convert DataC2 into DataA2'. There is no need to convert DataA1 and DataA2 since they are already in System A's data representation.

6. *Data assembly*: DataA1 and DataA1' are assembled by performing a union operation since the two sets of data answer the subquery to the same class. Similarly, DataA2 and DataA2' of subqueryA2 are assembled by performing a union operation. Finally, two sets of data that answer SubqueryA1 and

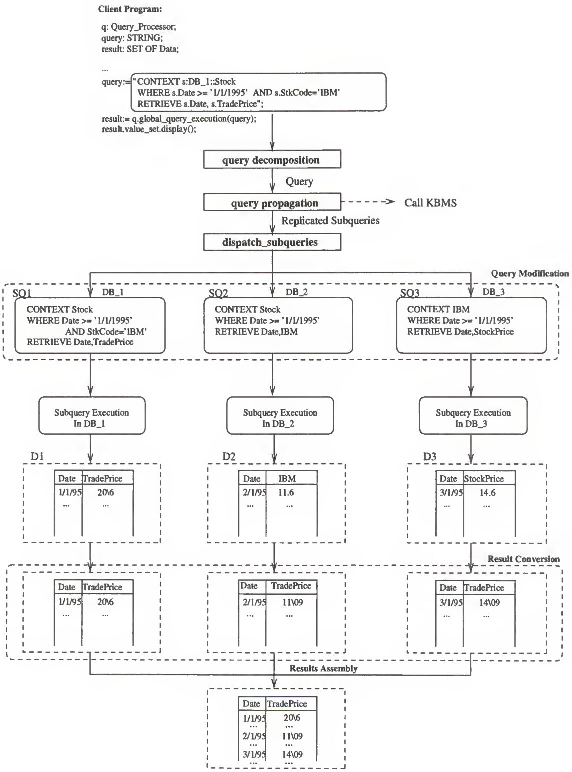
SubqueryA2 are joined based on the join conditions and association constraints specified in the original query.

7.3 An Example of Mediated Global Query Processing

Since the emphasis of this research is on the information mediation process rather than the distributed query processing, we shall use a simple query example which inquires data from multiple structurally different component systems as given in Section 4.3. This example will be sufficient to illustrate three major tasks of the mediation process, including locating information sources, subquery modification, and data conversion.

As shown in Figure 7.3, an OQL query for retrieving the stock data is issued based on the view of schema DB_1. The query is to retrieve all the stock trade price and the date information relating to the IBM company on or after the date, 1/1/1995. This query is passed as a parameter of a string value to the query processing method, global_query_execution().

Upon receiving the query, the DQP initializes, parses, checks and decomposes the query into simple subqueries, each of which accesses data from a single component system. Since the query given above is already simple, it is not decomposed. The query is stored in a data structure for initialization. After the query is initialized, rule_query_propagation is triggered, as described earlier, to access the mediation information from the KBMS. The meta-information of the global schema indicates that two other information sources (i.e., DB_2 and DB_3) contain relevant data. Thus, three instances are established in the entity class Query to store the original query



Client Program:

```
q: Query_Processor;
query: STRING;
result: SET OF Data;

...
query:="CONTEXT s:DB_1::Stock
        WHERE s.Date >= '1/1/1995'  AND s.StkCode='IBM'
        RETRIEVE s.Date, s.TradePrice";
result:= q.global_query_execution(query);
result.value_set.display();
```

SQ1 DB_1

```
CONTEXT Stock
WHERE Date >= '1/1/1995'
        AND StkCode='IBM'
RETRIEVE Date,TradePrice
```

SQ2 DB_2

```
CONTEXT Stock
WHERE Date >= '1/1/1995'
RETRIEVE Date,IBM
```

SQ3 DB_3

```
CONTEXT IBM
WHERE Date >= '1/1/1995'
RETRIEVE Date,StockPrice
```

D1

Date	TradePrice
1/1/95	20\6
...	...

D2

Date	IBM
2/1/95	11.6
...	...

D3

Date	StockPrice
3/1/95	14.6
...	...

Result Conversion

Date	TradePrice
1/1/95	20\6
...	...

Date	TradePrice
2/1/95	11\09
...	...

Date	TradePrice
3/1/95	14\09
...	...

Results Assembly

Date	TradePrice
1/1/95	20\6
...	...
2/1/95	11\09
...	...
3/1/95	14\09
...	...

Figure 7.3. Example of a Global Query Processing

and the schema identifiers of source and target systems. Each entry represents a replicated subquery to be dispatched to the SQP of a component system for local query processing. However, the two "replicated" subqueries are not ready for local processing since their representations are still based on the DB_1 schema. They need to be modified.

Rule_query_modification associated with the SQP of each component system is triggered before the three subqueries can be executed. By comparing the schema identifiers stored in the class Query, it detects that two of them (i.e., replicated for DB_2 and DB_3) need to be modified. After the execution of the query_modification method, the two subqueries are modified to be SQ2 and SQ3, as shown in Figure 7.3, which are represented in the views of schemas DB_2 and DB_3, respectively. Then all three subqueries are processed in their respective component systems. The returned data from DB_1, DB_2 and DB_3 are shown as D1, D2 and D3, respectively. As assumed previously, the representation of stock data in DB_1 is different from those in DB_2 and DB_3. Since the query is issued based on the view of schema DB_1, the returned data should conform to its representation. It requires some mediation operations to be performed to convert D2 and D3 into the New York Stock Exchange representation.

To achieve this, rule_data_conversion is triggered immediately after the stock data are retrieved from each component system. The method convert() is invoked to convert the data returned from DB_2 and DB_3 into the DB_1 data representation (i.e., the New York Stock Exchange representation). After the data conversion, the returned data are uniformly represented, as shown in the dotted rectangle labelled **Result Conversion** in Figure 7.3. The three sets of data are then assembled and the final result is returned to the user/application program which issued the query. A complete execution result of this query can be found in Appendix C.

CHAPTER 8
CONCLUSION AND FUTURE WORK

8.1 Conclusion

In this dissertation, we presented a mediation system that is based on a common modeling language, NCL, to resolve the data/information model differences, and a mediation specification language, MSL, to mediate the data heterogeneity problem. The two languages have been implemented and effectively achieve their individual objectives. At build-time, the component systems are uniformly modeled in NCL and their interrelationships are specified in MSL. The MSL specifications are then translated into the mediation elements in NCL specifications. The mediated global schema, combining the NCL specifications of component systems and mediation elements, is compiled to store the meta information in the KBMS and also generate enhanced CORBA program bindings and mediation code. The program bindings link different component systems (e.g., DQP, KBMS, clients, and SQPs and Mediators of information sources) together in a CORBA-based client/server distributed network environment.

At run-time, the client initiates a global query request by making a method call to DQP with an OQL query as the parameter. The DQP then checks and decomposes the global query into simple subqueries. Before subquery dispatching, a mediation rule is triggered to inquire the meta information from the KBMS and to locate information sources that contain relevant data. Based on the located information sources, more subqueries are propagated and then dispatched to the target systems for local processing. Before local query execution, a mediation rule is triggered to modify the query if a different view of data exists. After the local query execution, another mediation rule is triggered to convert data into the uniform representation specified in the original query. The DQP then collects and assembles the returned data, and returns the final result to the client.

Compared to other mediation research works, this research has the following distinct features.

1. *Resolving both the data/information model difference and the data heterogeneity problem in a unified framework*: This work combines and integrates the research efforts of two languages in a unified system. It provides a solution to the problem of both model and data heterogeneities found in the heterogeneous information system.

2. *Compilation approach to generate efficient code*: Unlike other mediation works using the interpreted logic rule inferencing approach, a compilation approach is taken to compile the mediation specification into program bindings and executables. It improves the query processing performance by distributing local query processing, rule code and mediation code to different component systems which perform query, rule and mediation processing.

3. *Schema mediation to allow users to see data in their own views*: Instead of performing schema integration, the mediation on component schemas preserves the views of local users. The users are able to issue global queries based on the views that they are already familiar with to retrieve data stored in other information sources.

4. *Supporting mediation system extensibility and scalability*: The mediation specification is based upon an O-O hierarchy which provides better extensibility and scalability. In this hierarchy, mediation specifications defined in lower levels can be reused to generate upper-level mediation specifications. It saves the efforts of creating and maintaining the mediation information. In addition, the cost of changing the mediation specification can be minimized since only the mediation specifications in an affected path of the hierarchy need to be updated. All the others remain the same.

5. *Distributed mediators to support localized mediation*: Instead of using a centralized mediator, the functions of mediation are distributed to the information sources. The mediation operations (i.e., query modification and data conversion) are carried out locally to avoid sending queries and data back and forth between the DQP and a centralized mediator.

Figure 8.1 shows the differences between our schema mediation approach and that of the TSIMMIS project [CHAW94] in the following five aspects: the means of forming mediation specification, user's views in issuing queries, computation approach, run-time query processing, and executions of mediation operations.

Projects / Compared Items	The Object-Oriented Mediated Schema	STANFORD TSIMMIS
Mediation Specification	Mediate the differnces between related object classes.	Mediate the differences between the integrated view and the underlying views.
User's view in issuing queries	Based on the mediated global schema. (i.e. users choose the view familiar to them.)	Based on the integrated views exported at different leves of mediation hierarchy (i.e., different levels of integrated views used by all users)
Computation approach	Compilation. Mediation specification is compiled to generate the specifications of DQP, Mediation classes, mediation rules and SQPs and Mediators of component systems Mediation specification is reused and compiled to generate code.	Interpretation. Mediation specification is interpreted at run-time to mediate the data heterogeneity.
Run-time query processing	Query passed directly to related information sources from the distributed query processor.	Query passed through intermediate mediators before reaching the information sources.
Mediation operations	Performed at the information sources	Performed at the sites of mediators

Figure 8.1. Differences between Our Mediation Approach and TSIMMIS'

8.2 Future Work

In this work, we have implemented a prototype mediation system which is tightly coupled with distributed query processing. The performance of the mediation system depends very much on the efficiency of distributed query processing which is not optimized at the present time. The mediation system can be extended in the following areas:

1. Query optimization can make use of the following information and approaches:

   - Local QP's capabilities: In our implementation of DQP, global queries are decomposed into simple subqueries (i.e., each involves only a single class) which are directly sent to the component systems for local processing.

However, in a heterogeneous information system, some component systems may have more query processing capabilities, such as performing multi-way JOINs. In that case, more complex subqueries can be formed for these component systems so that they can be processed concurrently instead of performing the joins in DQP.

- MEDIATION_TYPE constraint: The MEDIATION_TYPE clause in the mediation specification captures the interrelationships among a set of related classes. The relation operators, ONEOF, ANDOR and AND, indicate that the instances of these subclasses have Set-Exclusion, Set-Intersection and Set-Equality relationships, respectively. This meta information stored in the KBMS can be used to optimize the subquery generation. For example, if there exists a set-equality relationship between two subclasses of different component systems, the query can be sent to either one of them but not both since both classes contain the same set of instances.
- Network cost information: In a distributed information system, data transmissions and communications in the network are important performance factors. The time it takes to transmit data between each pair of component systems may differ from another pair. Also multiple component systems may contain the same data. If the DQP can select the component systems that can process subqueries and transmit data at a higher speed than others, the processing time of a global query can be greatly reduced.

Network transmission cost information can be stored and managed by the meta information manager (i.e., the KBMS) and be used by the DQP at run-time to select the proper component systems for query processing.

- Parallel execution of subqueries: The current implementation of DQP performs sequential subquery executions. To improve query processing performance, parallel processing of subqueries is necessary. This can be achieved by forking multiple concurrent processes or using the multithread facility provided by the CORBA communication infrastructure.

2. Distributed transaction management: Distributed query processing and mediation needs to be carried out in the framework of distributed transaction management. Issues related to concurrent control and error recovery also need to be dealt with in our future work.

3. Global semantic associations: In a heterogeneous information system, semantic associations among data entities across multiple component systems may exist. This information is not available in any of the component systems. They can be stored and managed by the KBMS.

4. Management of meta information: To support the use of local QP's capabilities and network cost information to do query optimization, the meta schema of the KBMS needs to be extended. Additional attributes for carrying the extra information need to be introduced and associated with the meta class, Schema.

With this extension, the KBMS will be able to provide the DQP with the useful information to improve its query processing performance at run-time.

APPENDIX A
BNF OF THE NIIIP COMMON LANGUAGE (NCL)

```
/* Starting Symbol: ncl_file */

ncl_file                :  defs
                        ;

defs                    :  /* NULL */
                        |  defs def
                        ;

def                     :  class_def
                        |  method_body_spec
                        |  global_rule_spec
                        |  program_def
                        ;

class_def               :  T_DEFINE class_type class_id  opt_equal_spec
                           opt_in_schema_spec
                           opt_class_specs
                           T_END_DEFINE T_SEMI
                        ;

class_type              :  T_CLASS_TYPE
                        ;

class_id                :  T_ID
                        ;


opt_equal_spec          :  /* NULL */
                        |  T_EQUAL underlying_type
                        ;
```

```
opt_in_schema_spec       :  /* NULL */
                         |  T_IN T_ID
                         ;

opt_class_specs          : /* NULL */
                         | class_specs
                         ;

class_specs              : subsuper T_SEMI
                           explicit_attr_list
                           derived_clause
                           inverse_clause
                           unique_clause
                           where_clause
                           assoc_section
                           method_section
                           local_rule_section
                           code_section
                         | T_SEMI constant_decl
                         ;


/* ------------ global rule specification --------------- */
global_rule_spec         :  rule_decls
                         ;


/* ------------ SUPERTYPE/SUBTYPE delcaration --------------- */
subsuper                  : /* NULL */
                          | supertype_decl subtype_decl
                          | subtype_decl supertype_decl
                          | subtype_decl
                          | supertype_decl
                          ;

supertype_decl            : T_ABSTRACT T_SUPERTYPE
                          | T_ABSTRACT T_SUPERTYPE T_OF
                            T_LEFT_PAREN
                            supertype_expr T_RIGHT_PAREN
                          | T_SUPERTYPE T_OF T_LEFT_PAREN
                            supertype_expr
                            T_RIGHT_PAREN
                          ;
```

```
supertype_expr            : supertype_factor
                          | supertype_expr T_AND supertype_factor
                          | supertype_expr T_ANDOR supertype_factor
                          ;

supertype_factor          : T_ID
                          | oneof
                          | T_LEFT_PAREN supertype_expr T_RIGHT_PAREN
                          ;

oneof                     : T_ONEOF
                            T_LEFT_PAREN supertype_list T_RIGHT_PAREN
                          ;

supertype_list            : supertype_expr
                          | supertype_list T_COMMA supertype_expr
                          ;

subtype_decl              : T_SUBTYPE T_OF  T_LEFT_PAREN
                            identifier_list T_RIGHT_PAREN
                          ;

identifier_list           : T_ID
                          | identifier_list T_COMMA T_ID
                          ;

/* ----------- explicit attribute delcarations -------------- */
explicit_attr_list        : /* NULL */
                          | explicit_attr_list explicit_attr
                          ;

explicit_attr             : attribute_list T_COLON opt
                            base_type opt_where_cons T_SEMI
                          ;

opt_where_cons           : /* NULL */
                         | T_WHERE T_LEFT_PAREN add_cons T_RIGHT_PAREN
                         ;

add_cons                 : T_CONS_ID opt_para
                         | add_cons T_COMMA T_CONS_ID opt_para
                         ;

opt_para                 : /* NULL */
```

```
                                | T_LEFT_PAREN bound_colon_list T_RIGHT_PAREN
                                | expr
                                ;


attribute_list                   : attribute_decl
                                 | attribute_list T_COMMA attribute_decl
                                 ;

attribute_decl                   : T_ID
                                 | qualified_attribute
                                 ;

qualified_attribute              : T_SELF
                                   group_qualifier attribute_qualifier
                                 ;

attribute_qualifier              : T_DOT T_ID
                                 ;

group_qualifier                  : T_BACKSLASH T_ID
                                 ;

opt                              : /* NULL */
                                 | T_OPTIONAL
                                 ;

bound_spec                       : T_LEFT_BRACKET  bound_1 T_COLON
                                   bound_2 T_RIGHT_BRACKET
                                 ;

bound_1                          : expr
                                 ;

bound_2                          : expr
                                 ;

expr                             : expr_list_expr
                                 | binary_operation
                                 | unary_operation
                                 | list_expr
                                 | T_CONTEXT context_expr
                                 | existential_quant
                                 | universal_quant
                                | parenthesis_expr
                                | bracket_expr
                                | dotted_expr
                                | single_item
                                | interval_expr
                                | query_expr
                                ;

/* exp { exp,exp,...} */
expr_list_expr                  : expr list_expr
                                ;

list_expr                       : T_LEFT_CURL expr_list T_RIGHT_CURL
                                ;

expr_list                       : /* NULL */
                                | expr_ls
                                ;

expr_ls                         : expr
                                | expr_ls T_COMMA expr
                                ;

/* exp <op> exp */
binary_operation                : expr T_ASSIGNMENT expr
                                | binary1_operation
                                | binary2_operation
                                | binary3_operation
                                ;

binary1_operation               : expr T_LESS_THAN expr
                                | expr T_LESS_EQUAL expr
                                | expr T_GREATER_THAN expr
                                | expr T_GREATER_EQUAL expr
                                | expr T_NOT expr
                                | expr T_EQUAL expr
                                | expr T_NOT_EQUAL expr
                                | expr T_IN expr
                                | expr T_LIKE expr
                                | expr T_INST_NOT_EQUAL expr
                                | expr T_INST_EQUAL expr
                                ;

binary2_operation               : expr T_OR expr
```

```
                        | expr T_PLUS expr
                        | expr T_MINUS expr
                        | expr T_XOR expr
                        ;

binary3_operation       : expr T_TIMES  expr
                        | expr T_DIV expr
                        | expr T_MOD expr
                        | expr T_AND expr
                        | expr T_DOLLAR expr
                        | expr T_REAL_DIV expr
                        | expr T_CONCAT_OP expr
                        | expr T_EXP expr
                        ;

/* <op> expr */
unary_operation       : T_LESS_THAN expr
                      | T_LESS_EQUAL expr
                      | T_GREATER_THAN expr
                      | T_GREATER_EQUAL expr
                      | T_NOT expr
                      | T_EQUAL expr
                      | T_NOT_EQUAL expr
                      | T_OR expr
                      | T_PLUS expr
                      | T_MINUS expr
                      | T_TIMES expr
                      | T_DIV expr
                      | T_MOD expr
                      | T_AND expr
                      | T_AMPERSAND expr
                      ;

context_expr          : pattern_spec opt_where_clause opt_select_clause
                      ;

pattern_spec          : context_expr_list
                      | binary_context
                      | T_LEFT_PAREN pattern_spec T_RIGHT_PAREN
                      | single_context
                      ;

context_expr_list     : T_LEFT_CURL pattern_list T_RIGHT_CURL
                      ;
```

```
pattern_list          : pattern_list_spec
                      | pattern_list T_COMMA pattern_list_spec
                      ;

pattern_list_spec     : binary_context
                      | unary_context
                      | single_context
                      ;

binary_context        : binary1_context
                      | binary2_context
                      | binary3_context
                      ;

binary1_context       : pattern_spec T_AND context_expr_list
                      | pattern_spec T_OR context_expr_list
                      ;

binary2_context       : pattern_spec T_BANG opt_link pattern_spec
                      | pattern_spec T_NASSOC_L opt_link pattern_spec
                      | pattern_spec T_NASSOC_R opt_link pattern_spec
                      ;


opt_link              :  /*EMPTY*/
                      | T_LEFT_BRACKET T_ID T_RIGHT_BRACKET
                      ;

binary3_context       :  pattern_spec T_TIMES opt_link pattern_spec
                      |  pattern_spec T_ASSOC_L opt_link pattern_spec
                      |  pattern_spec T_ASSOC_R opt_link pattern_spec
                      ;

unary_context         :  T_TIMES opt_link pattern_spec
                      |  T_ASSOC_L opt_link pattern_spec
                      |  T_ASSOC_R opt_link pattern_spec
                      |  T_BANG opt_link pattern_spec
                      |  T_NASSOC_L opt_link pattern_spec
                      |  T_NASSOC_R opt_link pattern_spec
                      ;

single_context        :  T_ID
                      |  T_ID T_COLON T_ID
```

```
                        | T_ID T_TWO_COLON T_ID
                        | T_ID T_COLON T_ID T_TWO_COLON  T_ID
                        ;

opt_where_clause        : /*EMPTY*/
                        | T_WHERE expr
                        ;

opt_select_clause       : /*EMPTY*/
                        | T_SELECT identifier_list
                        ;

variable_list           : identifier_list
                        ;

/* EXIST ID,ID,... IN <context_exp> */
existential_quant       : T_EXIST variable_list T_IN context_expr
                        ;

/* FORALL ID,ID,... IN <context_exp> SUCHTHAT <exp> */
universal_quant         : T_FORALL variable_list T_IN context_expr
                          T_SUCHTHAT expr
                        ;

/* ( <exp> ) */
parenthesis_expr        : T_LEFT_PAREN
                          expr T_RIGHT_PAREN
                        ;

/* [ <exp> ] */
bracket_expr            : expr brackets
                        ;

brackets                : T_LEFT_BRACKET
                          expr T_RIGHT_BRACKET
                        ;

/* <exp> . ID or <exp>.method_call */
dotted_expr             : expr T_DOT T_ID
                        | expr T_DOT method_invocation
                        ;

/* { <exp> <= <exp> <= <exp> } */
interval_expr           :  T_LEFT_CURL expr interval_op
```

```
                           expr interval_op
                           expr  T_RIGHT_CURL
                         ;

interval_op              :  T_LESS_EQUAL
                         |  T_LESS_THAN
                         ;

/* QUERY ( ID <* <exp> | <exp> ) */
query_expr               :  T_QUERY T_LEFT_PAREN qualified_name
                            T_Q_STAR expr T_BAR
                            expr T_RIGHT_PAREN
                         ;

/* single item (to terminals) */
single_item              : method_invocation
                         | qualified_name
                         | literal
                         | qualifiable_factor  qualifier_list
                         ;

literal                  :  T_BINARY_LITERAL
                         |  T_INTEGER_LITERAL
                         |  T_LOGICAL_LITERAL
                         |  T_REAL_LITERAL
                         |  T_STRING_LITERAL
                         |  T_ENCODED_STRING_LITERAL
                         ;

qualifiable_factor       :  constant_factor
                         | T_ID qualifier
                         ;

constant_factor           :  built_in_constant
                          ;

built_in_constant         :  T_CONST_E
                          |  T_PI
                          |  T_SELF
                          |  T_QUESTIONMARK
                          |  T_NULL        /* appended */
                          ;

method_invocation         :  T_BUILTIN_METHOD
```

```
                            T_LEFT_PAREN
                            expr_list T_RIGHT_PAREN
                         |  T_ID
                            T_LEFT_PAREN
                            expr_list T_RIGHT_PAREN
                         ;

qualifier_list           :  /* NULL */
                         |  qualifier_list qualifier
                         ;

qualifier                :  attribute_qualifier
                         |  group_qualifier
                         |  subcomponent_qualifier
                         ;

subcomponent_qualifier   :  T_LEFT_BRACKET expr
                            T_RIGHT_BRACKET
                         |  T_LEFT_BRACKET  expr T_COLON
                            expr T_RIGHT_BRACKET
                         ;

base_type                :  aggregation_types
                         |  simple_types
                         |  qualified_name
                         ;

aggregation_types        :  array_type
                         |  bag_type
                         |  set_type
                         |  list_type
                         ;

array_type               :  T_ARRAY
                            bound_spec T_OF opt unique base_type
                         ;

unique                   :  /* NULL */
                         |  T_UNIQUE
                         ;

bag_type                 :  T_BAG gen_bound_spec T_OF base_type
                         ;
```

```
gen_bound_spec          :  bound_spec
                        |  /* NULL */
                        ;

set_type                :  T_SET
                           gen_bound_spec  T_OF base_type
                        ;

list_type               :  T_LIST
                           gen_bound_spec T_OF unique base_type
                        ;

simple_types            :  T_BINARY precision_spec opt_fixed
                        |  T_BOOLEAN
                        |  T_INTEGER precision_spec
                        |  T_LOGICAL
                        |  T_NUMBER
                        |  T_REAL precision_spec
                        |  T_STRING precision_spec opt_fixed
                        |  T_VOID
                        ;

precision_spec          :  /* NULL */
                        |  T_LEFT_PAREN expr T_RIGHT_PAREN
                        ;


opt_fixed               :  /* NULL */
                        |  T_FIXED
                        ;

/* ----------- DERIVE clause -------------- */
derived_clause          :  T_DERIVE  derived_attr_list
                        |  /* NULL */
                        ;

derived_attr_list       :  derived_attr
                        |  derived_attr_list derived_attr
                        ;

derived_attr            :  attribute_decl  T_COLON  base_type
                           T_ASSIGNMENT expr T_SEMI
                        ;
```

```
/* ------------ INVERSE clause --------------- */
inverse_attr            :  T_ID
                           T_COLON inverse_type
                           T_FOR T_ID T_SEMI
                        ;


inverse_attr_list       :  inverse_attr
                        |  inverse_attr_list inverse_attr
                        ;

inverse_clause          :  T_INVERSE  inverse_attr_list
                        |  /* NULL */
                        ;

inverse_type            :  set_or_bag T_ID
                        ;

set_or_bag              :  T_SET gen_bound_spec T_OF
                        |  T_BAG gen_bound_spec T_OF
                        |  /* NULL */
                        ;

/* ------------ UNIQUE clause --------------- */
unique_clause           :  T_UNIQUE  unique_rule_list
                        |  /* NULL */
                        ;

unique_rule_list        :  unique_rule
                        |  unique_rule_list   unique_rule
                        ;

unique_rule             :  attribute_list T_SEMI
                        |  T_ID
                           T_COLON attribute_list
                           T_SEMI
                        ;

/* ------------ WHERE (domain rule) clause --------------- */
where_clause            :  T_WHERE
                           domain_rule_list
                        |  /* NULL */
                        ;
```

```
domain_rule_list         :  domain_rule
                         |  domain_rule_list   domain_rule
                         ;

domain_rule              :  T_ID T_COLON expr
                            T_SEMI
                         |  expr T_SEMI
                         ;

/* ------------ ASSOCIATION section --------------- */
assoc_section            :  T_ASSOCIATIONS T_COLON assoc_specs
                         |  /* NULL */
                         ;

assoc_specs              :  assoc_spec
                         |  assoc_specs assoc_spec
                         ;

assoc_spec               :  assoc_type opt_to_of
                            attr_or_param_list
                            opt_card_output T_SEMI
                         ;

assoc_type               :  T_ASSOC_TYPE
                         ;

opt_to_of                : /* NULL */
                         | T_TO
                         | T_OF
                         ;

attr_or_param_list       :  T_LEFT_PAREN attr_or_param_rep T_RIGHT_PAREN
                         ;

attr_or_param_rep        :  T_ID
                         |  formal_param_rep
                         ;

opt_card_output          : /* NULL */
                         | output_clause
                         | card_clause
                         ;
```

```
card_clause               : T_CARDINALITY T_LEFT_PAREN sub_card_list
                            T_RIGHT_PAREN
                          ;

sub_card_list             :  sub_card
                          |  sub_card_list T_SEMI  sub_card
                          ;

sub_card                  :  attr_colon_list T_EQUAL bound_colon_list T_SEMI
                          ;

attr_colon_list           :  T_ID
                             T_COLON T_ID
                          ;

bound_colon_list          :  bound_spec
                             T_COLON bound_spec
                          ;


output_clause             : T_OUTPUT_DATA T_LEFT_PAREN T_ID T_RIGHT_PAREN
                          ;


/* ----------- METHOD declaration section -------------- */
method_section            :  T_METHODS T_COLON
                             exception_list  method_list
                          |  /* NULL */
                          ;

exception_list            :  /* NULL */
                          |  exception_list exception_decl
                          ;

exception_decl            :  T_EXCEPTION T_ID
                             formal_param_list T_SEMI
                          ;

formal_param_list         :  /* NULL */
                          |  T_LEFT_PAREN formal_param_rep T_RIGHT_PAREN
                          ;

formal_param_rep          :  formal_param
                          |  formal_param_rep T_SEMI formal_param
```

```
                         ;

formal_param             :  attribute_list T_COLON para_type
                         ;

method_list              :  method_decl
                         |  method_list method_decl
                         ;

method_decl              :  T_METHOD  oneway  T_ID
                            method_params
                            T_COLON return_type
                            raise
                            T_SEMI
                         ;

oneway                   :  /* NULL */
                         |  T_ONEWAY
                         ;

method_params            :  T_LEFT_PAREN T_RIGHT_PAREN
                         |  T_LEFT_PAREN method_parameter_rep T_RIGHT_PAREN
                         ;

method_parameter_rep     :  method_parameter
                         |  method_parameter_rep T_SEMI method_parameter
                         ;

method_parameter         :  passing_type  attribute_list T_COLON para_type
                         ;


passing_type             : T_IN
                         | T_OUT
                         | T_INOUT
                         ;

para_type                :  generalized_types
                         |  simple_types
                         |  qualified_name
                         ;

generalized_types        :  aggregate_type
                         |  general_aggregation_types
                         |  generic_type
                         ;

aggregate_type           :  T_AGGREGATE T_OF para_type
                         |  T_AGGREGATE T_COLON T_ID T_OF para_type
                         ;

general_aggregation_types:  general_array_type
                         |  general_bag_type
                         |  general_set_type
                         |  general_list_type
                         ;

general_array_type       :  T_ARRAY  gen_bound_spec T_OF para_type
                         ;

general_bag_type         :  T_BAG gen_bound_spec T_OF para_type
                         ;

general_list_type        :  T_LIST gen_bound_spec T_OF para_type
                         ;

general_set_type         :  T_SET gen_bound_spec T_OF para_type
                         ;

generic_type             :  T_GENERIC
                         |  T_GENERIC T_COLON T_ID
                         ;

return_type              : T_VOID
                         | para_type
                         ;

raise                    : /* NULL */
                         | T_RAISES exception_id_list
                         ;

exception_id_list        : T_LEFT_PAREN identifier_list T_RIGHT_PAREN
                         ;

/* ----------- local RULE section -------------- */
local_rule_section       : /* NULL */
                         | T_RULES T_COLON rule_decls
```

```
                          ;

rule_decls                : rule_decl
                          | rule_decls rule_decl
                          ;

rule_decl                 : rule_head opt_rule_trigger rule_body
                            T_END_RULE T_SEMI
                          ;

rule_head                 : T_RULE T_ID T_SEMI
                          ;

opt_rule_trigger          : /* NULL */
                          | rule_trigger
                          ;

rule_trigger              : T_TRIGGERED trigger_conds
                          ;

trigger_conds             : trigger_cond
                          | trigger_conds trigger_cond
                          ;

trigger_cond              : trigger_time  operation_list
                          ;

trigger_time              : T_BEFORE
                          | T_AFTER
                          | T_IMMED_AFTER
                          ;

operation_list            : operation_spec
                          | operation_list T_COMMA operation_spec
                          ;

operation_spec            : qualified_name T_LEFT_PAREN  opt_param_decls
                            T_RIGHT_PAREN
                          ;

qualified_name            : T_ID
                          | T_ID T_TWO_COLON T_ID
                          ;
```

```
opt_param_decls          : /* NULL */
                         | param_decls
                         ;

param_decls              : param_spec
                         | param_decls T_COMMA param_spec
                         ;

param_spec               : T_ID T_COLON class_spec
                         | class_spec
                         ;

class_spec               : base_type
                         ;

rule_body                : opt_rule_cond opt_rule_action opt_rule_alt
                         ;

opt_rule_cond            : /* NULL */
                         | rule_cond_spec
                         ;

rule_cond_spec           : T_CONDITION rule_cond
                         ;

rule_cond                : expr
                         | expr T_BAR expr
                         ;

opt_rule_action          : /* NULL */
                         | rule_action
                         ;

rule_action              : T_ACTION  stmts
                         ;

opt_rule_alt             : /* NULL */
                         | rule_alt
                         ;

rule_alt                 : T_OTHERWISE stmts
                         ;
```

```
/* ------------ executable statements --------------- */
stmts                   : stmt
                        | stmts stmt
                        ;

stmt                    : expr T_SEMI
                        | null_stmt
                        | branch_stmt T_SEMI
                        | loop_stmt T_SEMI
                        | flow_stmt T_SEMI
                        | local_stmt T_SEMI
                        | expr T_DO stmts T_END T_SEMI
                        | T_BEGIN stmts T_END T_SEMI
                        ;

null_stmt               : T_SEMI
                        ;

branch_stmt             : if_stmt
                        | case_stmt
                        ;

if_stmt                 : T_IF  expr
                          T_THEN stmts
                          opt_else T_END_IF
                        ;

opt_else                : /* NULL */
                        | T_ELSE  stmts
                        ;

/* confer with case_stmt in EXPRESS */
case_stmt               : T_CASE  multiple_when
                          opt_otherwise
                          T_END_CASE
                        ;

multiple_when           : when_stmt
                        | multiple_when when_stmt
                        ;

when_stmt               : T_WHEN expr T_DO stmt
                        ;
```

```
opt_otherwise            : /* NULL */
                         | T_OTHERWISE T_DO stmt
                         ;

loop_stmt                : for_stmt
                         | while_stmt
                         ;

for_stmt                 : T_FOR expr T_UNTIL expr T_BY expr
                           T_DO stmts T_END_FOR
                         ;

while_stmt               : T_WHILE expr T_DO
                           stmts T_END_WHILE
                         ;

flow_stmt                : T_BREAK
                         | T_CONTINUE
                         | T_ABORT
                         | return_stmt
                         ;

return_stmt              : T_RETURN opt_expr
                         ;

opt_expr                 : /* NULL */
                         | expr
                         ;

/* ------------ local variable decls -------------- */
opt_local                : /* NULL */
                         | local_stmt
                         ;

local_stmt               : T_LOCAL
                           var_decls T_END_LOCAL
                           T_SEMI
                         ;

var_decls                : var_decl
                         | var_decls var_decl
                         ;

var_decl                 : attribute_list
```

```
                          T_COLON class_spec T_SEMI
                         ;

/* ------------ method body spec --------------- */
method_body_spec         : method_head
                          opt_local
                          stmts
                          T_END_METHOD  T_SEMI
                         ;

method_head              :  T_METHOD qualified_name  T_LEFT_PAREN
                            T_RIGHT_PAREN T_SEMI
                         ;


/* ------------ about data types  --------------- */
underlying_type          :  constructed_types
                         |  aggregation_types
                         |  simple_types
                         |  T_ID
                         ;

constructed_types        :  enumeration_type
                         |  select_type
                         ;

enumeration_type         :  T_ENUMERATION T_OF
                            T_LEFT_PAREN identifier_list T_RIGHT_PAREN
                         ;

select_type              :  T_SELECT T_OF
                            T_LEFT_PAREN identifier_list T_RIGHT_PAREN
                         ;

/* ------------ code section --------------- */
code_section             : /* NULL */
                         | T_CODES T_COLON code_list
                         ;

code_list                : text_block
                         | var_code_list
                         ;

text_block               :  T_BEGIN_TEXT stmts T_END_TEXT
```

```
                        ;

var_code_list           : var_code T_SEMI
                        | var_code_list var_code T_SEMI
                        ;

var_code                : T_ID
                          T_ASSIGNMENT code_expression
                        ;

code_expression         : text_block
                        | expr
                        ;

/* ------------ constant_decl  --------------- */
constant_decl           :  T_CONSTANT constant_list T_END_CONSTANT
                           T_SEMI
                        ;

constant_list           :  constant_body
                        |  constant_list constant_body
                        ;

constant_body           :  T_ID T_COLON base_type T_ASSIGNMENT
                           expr T_SEMI
                        ;


/* ------------ program_def  --------------- */

program_def             :  T_PGM T_ID  opt_local stmts
                           T_END_PGM T_SEMI
                        ;
```

APPENDIX B
BNF OF THE MEDIATION SPECIFICATION LANGUAGE (MSL)

```
/* Starting Symbol: mediation_file */

mediation_file           :  /* NULL */
                         |  mediation_file schema_decl
                         ;

schema_decl              :  schema_header
                            schema_body TOK_END_SCHEMA semicolon
                         ;

schema_header            :  TOK_SCHEMA TOK_IDENTIFIER semicolon
                         ;

schema_body              :  interface_spec_list schema_block_list
                         ;

interface_spec_list      :  /* NULL no interface specification */
                         |  interface_spec_list interface_specification
                         ;

interface_specification :  use_clause
                         ;

use_clause               :  TOK_USE TOK_FROM TOK_IDENTIFIER semicolon
                         |  TOK_USE TOK_FROM TOK_IDENTIFIER
                            TOK_LEFT_PAREN entity_id_list
                            TOK_RIGHT_PAREN semicolon
                         ;

entity_id_list           :  entity_id
                         |  entity_id_list comma entity_id
                         ;
```

```
entity_id               :  TOK_IDENTIFIER;

schema_block_list       :  /* NULL */
                        |  schema_block_list schema_block
                        ;

schema_block            :  declaration
                        ;

declaration             :  entity_decl
                        ;

entity_decl             :  entity_head
                           supertype_declaration semicolon
                           entity_body TOK_END_ENTITY semicolon
                        ;

entity_head             :  TOK_ENTITY TOK_IDENTIFIER
                        ;

supertype_declaration   :  TOK_ABSTRACT TOK_SUPERTYPE
                        |  TOK_ABSTRACT TOK_SUPERTYPE  TOK_OF
                           TOK_LEFT_PAREN  supertype_expression
                           TOK_RIGHT_PAREN
                        ;

supertype_expression    :  supertype_factor
                        |  supertype_expression TOK_AND  supertype_factor
                        |  supertype_expression TOK_ANDOR  supertype_factor
                        ;

supertype_factor        :  super_sch_entity
                        |  oneof
                        |  TOK_LEFT_PAREN supertype_expression TOK_RIGHT_PAREN
                        ;

super_sch_entity        :  TOK_IDENTIFIER T_TWO_COLON TOK_IDENTIFIER
                        ;

oneof                   :  TOK_ONEOF
                           TOK_LEFT_PAREN supertype_list TOK_RIGHT_PAREN
                        ;

supertype_list          :  supertype_expression
```

```
                        |  supertype_list TOK_COMMA supertype_expression
                        ;

entity_body             :  entity_equ_clause attr_equ_list
                        ;

entity_equ_clause       : TOK_ENTITY TOK_EQUIVALENCE TOK_LEFT_PAREN
                          sch_entity_list
                          TOK_RIGHT_PAREN semicolon
                        ;

sch_entity_list         : sch_entity comma sch_entity
                        | sch_entity_list comma sch_entity
                        ;

sch_entity              : schema_id T_TWO_COLON entity_id
                        ;

schema_id               : TOK_IDENTIFIER
                        ;

entity_id               :  TOK_IDENTIFIER
                        ;

attr_equ_list           : attr_equ_clause
                        | attr_equ_list attr_equ_clause
                        ;

attr_equ_clause         : TOK_ATTRIBUTE TOK_EQUIVALENCE
                          TOK_LEFT_PAREN
                          sch_entity_attr_list
                          TOK_RIGHT_PAREN semicolon
                          opt_value_equ_clause
                          opt_where_clause
                        ;

sch_entity_attr_list    : s_e_a_element comma s_e_a_element
                        | sch_entity_attr_list comma s_e_a_element
                        ;

s_e_a_element           : sch_entity_attr
                        | TOK_LEFT_PAREN s_e_a_list TOK_RIGHT_PAREN
                        ;
```

```
s_e_a_list              : sch_entity_attr
                        | s_e_a_list comma sch_entity_attr
                        ;

sch_entity_attr         : schema_id T_TWO_COLON entity_id TOK_DOT attr_id
                        | schema_id T_TWO_COLON entity_id
                          TOK_BACKSLASH schema_id T_TWO_COLON entity_id
                          TOK_DOT attr_id
                        ;

attr_id                 : TOK_IDENTIFIER
                        ;

opt_value_equ_clause    : /* NULL */
                        | value_equ_clause
                        ;

value_equ_clause        : TOK_VALUE TOK_EQUIVALENCE
                          TOK_LEFT_PAREN value_equ_para_list
                          TOK_RIGHT_PAREN semicolon
                        ;

value_equ_para_list     : value_equ_para
                        | value_equ_para_list semicolon value_equ_para
                        ;

value_equ_para          : value_equ_type2
                        | /* NULL */
                        ;

value_equ_type2         : TOK_LEFT_PAREN sch_entity_attr
                          comma attr_conv_method_list TOK_RIGHT_PAREN
                        ;

attr_conv_method_list   : attr_conv_method
                        | attr_conv_method_list comma attr_conv_method
                        ;

attr_conv_method        : conv_method_id
                          TOK_LEFT_PAREN s_e_a_list
                          TOK_RIGHT_PAREN
                        ;

conv_method_id          : TOK_IDENTIFIER
```

```
                        ;

opt_where_clause        : /* NULL */
                        | where_clause
                        ;

where_clause            :  TOK_WHERE
                           domain_rule_list
                        ;

domain_rule_list        :  domain_rule
                        |  domain_rule_list domain_rule
                        ;

domain_rule             :  expression semicolon
                        ;

expression              :  simple_expression
                        |  expression rel_op_extended simple_expression
                        ;

rel_op_extended         :  rel_op
                        ;

rel_op                  :  TOK_LESS_THAN
                        |  TOK_GREATER_THAN
                        |  TOK_LESS_EQUAL
                        |  TOK_GREATER_EQUAL
                        |  TOK_NOT_EQUAL
                        |  TOK_EQUAL
                        |  TOK_INST_NOT_EQUAL
                        |  TOK_INST_EQUAL
                        ;

simple_expression       :  term
                        |  simple_expression add_like_op  term
                        ;

add_like_op             :  TOK_PLUS
                        |  TOK_MINUS
                        |  TOK_OR
                        |  TOK_XOR
                        ;
```

```
term                    :  factor
                        |  term multiplication_like_op  factor
                        ;

multiplication_like_op  :  TOK_TIMES
                        |  TOK_REAL_DIV
                        |  TOK_CONCAT_OP
                        |  TOK_AND
                        |  TOK_DIV
                        |  TOK_MOD
                        ;

factor                  :  simple_factor TOK_EXP simple_factor
                        |  simple_factor
                        ;

simple_factor           :  TOK_NOT TOK_LEFT_PAREN
                           expression TOK_RIGHT_PAREN
                        |  TOK_LEFT_PAREN
                           expression TOK_RIGHT_PAREN
                        |  unary_op primary
                        |  primary
                        ;

unary_op                :  TOK_PLUS
                        |  TOK_MINUS
                        |  TOK_NOT
                        ;

primary                 :  literal
                        |  identifier
                        ;

literal                 :  TOK_BINARY_LITERAL
                        |  TOK_INTEGER_LITERAL
                        |  TOK_LOGICAL_LITERAL
                        |  TOK_REAL_LITERAL
                        |  TOK_STRING_LITERAL
                        ;

identifier              :  token qualifier_list
                        ;

token                   :  TOK_IDENTIFIER
```

```
                         |  TOK_SELF TOK_DOT TOK_IDENTIFIER
                         ;

qualifier_list           :  /* NULL */
                         |  qualifier_list qualifier
                         ;

qualifier                :  attribute_qualifier
                         |  group_qualifier
                         |  class_qualifier
                         ;

attribute_qualifier      :  TOK_DOT TOK_IDENTIFIER
                         ;

group_qualifier          :  TOK_BACKSLASH TOK_IDENTIFIER
                         ;

class_qualifier          : T_TWO_COLON TOK_IDENTIFIER
                         ;
```

APPENDIX C
RESULT OF A GLOBAL QUERY EXECUTION

Complete execution results of the query:

```
CONTEXT s:DB1::Stock
WHERE s.Date >= '1/1/1995' AND s.StkCode = 'IBM'
RETRIEVE (s.Date, s.TradePrice);


CLIENT: Sending a query to DQP

DQP: Query received.
  context DB_1::Stock
  where (DB_1::Stock.Date>"1/1/95")and(DB_1::Stock1.StkCode="IBM")
  retrieve (DB_1::Stock.Date,DB_1::Stock1.TradePrice);

DQP: One subquery is generated.

KBMS: Triggered by a mediation rule
  3 information source(s), DB_1, DB_2 and DB_3, have been located.

DQP: Send subquery to DB_1_SQP

DB_1_SQP: Query received
  context DB_1::Stock
  where (DB_1::Stock.Date>"1/1/95")and(DB_1::Stock1.StkCode="IBM")
  retrieve (DB_1::Stock.Date,DB_1::Stock1.TradePrice);
  TARGET SYSTEM:DB_1;

DB_1_SQP: No change necessary

DB_1_SQP: Processing query in DB_1
 DB_1::Stock.Date DB_1::Stock1.TradePrice
         1/1/95          20\6
         1/2/95          21\5
         1/3/95          22\3
         1/4/95          23\1
```

```
DB_1_Mediator: Data conversion begins.
  DB_1::Stock.Date: No conversion needed

  DB_1::Stock.TradePrice: No conversion needed

DQP: Data received from DB_1_SQP (4 instances)
DB_1::Stock.Date DB_1::Stock1.TradePrice
-------------------------------------------------------------
           1/1/95                    20\6
           1/2/95                    21\5
           1/3/95                    22\3
           1/4/95                    23\1

DQP: Send subquery to DB_2_SQP

DB_2_SQP: Query received
  context DB_1::Stock
  where (DB_1::Stock.Date>"1/1/95")and(DB_1::Stock1.StkCode="IBM")
  retrieve (DB_1::Stock.Date,DB_1::Stock1.TradePrice);
  TARGET SYSTEM:DB_2;

DB_2_SQP: Query needs to be modified.

DB_2_Mediator: Triggered to modify the query

DB_2_SQP: Modified query
  context DB_2::Stock
  where (DB_2::Stock.Date>"1/1/95")
  retrieve (DB_2::Stock.Date,DB_2::Stock2.IBM);
  TARGET SYSTEM:DB_2;

DB_2_SQP: Processing query in DB_2
 DB_2::Stock.Date DB_2::Stock2.IBM
          2/1/95        11.6
          2/2/95        11.5
          2/3/95        12.3
          2/4/95        13

DB_2_Mediator: Data conversion begins.
  DB_2::Stock.Date -> DB_1::Stock.Date

  DB_2::Stock.IBM -> DB_1::Stock.TradePrice
    11.6 -> 11\09
    11.5 -> 11\08
    12.3 -> 12\04
    13 -> 13\00

DQP: Data received from DB_2_SQP (4 instances)
DB_1::Stock.Date DB_1::Stock1.TradePrice
-------------------------------------------------------------
           2/1/95                   11\09
           2/2/95                   11\08
           2/3/95                   12\04
           2/4/95                   13\00

DQP: Send subquery to DB_3_SQP

DB_3_SQP: Query received
  context DB_1::Stock
  where (DB_1::Stock.Date>"1/1/95")and(DB_1::Stock1.StkCode="IBM")
  retrieve (DB_1::Stock.Date,DB_1::Stock1.TradePrice);
  TARGET SYSTEM:DB_3;

DB_3_SQP: Query needs to be modified.

DB_3_Mediator: Triggered to modify the query

DB_3_SQP: Modified query
  context DB_3::IBM
  where (DB_3::IBM.Date>"1/1/95")
  retrieve (DB_3::IBM.Date,DB_3::IBM.StockPrice);
  TARGET SYSTEM:DB_3;

DB_3_SQP: Processing query in DB_3
 DB_3::IBM.Date DB_3::IBM.StockPrice
          3/1/95         14.6
          3/2/95         15.5
          3/3/95         14.3
          3/4/95         16

DB_3_Mediator: Data conversion begins.
  DB_3::IBM.Date -> DB_1::Stock.Date

  DB_3::IBM.StockPrice -> DB_1::Stock.TradePrice
    14.6 -> 14\09
    15.5 -> 15\08
    14.3 -> 14\04
    16 -> 16\00

DQP: Data received from DB_3_SQP (4 instances)
DB_1::Stock.Date DB_1::Stock1.TradePrice
------------------------------------------------------------
           3/1/95                      14\09
           3/2/95                      15\08
           3/3/95                      14\04
           3/4/95                      16\00

DQP: Assemble data of the 3 subqueries.

DQP: Return data to CLIENT

CLIENT: Receiving data from DQP
DB_1::Stock.Date DB_1::Stock1.TradePrice
------------------------------------------------------------
           1/1/95                       20\6
           1/2/95                       21\5
           1/3/95                       22\3
           1/4/95                       23\1
           2/1/95                      11\09
           2/2/95                      11\08
           2/3/95                      12\04
           2/4/95                      13\00
           3/1/95                      14\09
           3/2/95                      15\08
           3/3/95                      14\04
           3/4/95                      16\00

END
```

REFERENCES

[ALA89] A. Alashqur, S. Y. W. Su and H. Lam, "OQL– A Query Language for Manipulating Object-oriented Databases," *Proceedings of 15th International Conference on Very Large Data Bases*, Amsterdam, Netherlands, 1989, pp. 433-442.

[AMB94] J. L. Ambite, Y. Arens, C. Chee, C. N. Hsu, and C. A. Knoblock, "SIMS Manual," Working Draft, July 1994.

[ARR97] J. A. Arroyo-Figueroa, "An Extensible Knowledge Base Programming Language and its Extensible Object Model," Ph.D. Dissertation, University of Florida, May 1997.

[BRE90] Y. Breitbart, "Multidatabase Interoperability," *SIGMOD Record*, Vol. 19, No. 3, September 1990, pp. 53-60.

[CHAL94] H. Chalupsky and S. C. Shapiro, "Ontological Mediation: An Analysis," Draft, Department of Computer Science, State University of New York at Buffalo, Buffalo, NY, July 1994.

[CHAT91] A. Chatterjee and A. Segev, "Data Manipulation in Heterogeneous Databases," *SIGMOD Record*, Vol. 20, No. 4, December 1991, pp. 64-68.

[CHAW94] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom, "The TSIMMIS Project: Integration of Heterogeneous Information Sources," *Proceedings of IPSJ Conference*, Tokyo, Japan, October 1994, pp. 7-18.

[CHE88] L. Chen and P. Arbee, "Schema Integration to Support Multiple Database Access," Bellcore Technical Report TM-STS-012948, December 1988.

[COL91] C. Collet, M. N. Huhns and W.-M. Shen, "Resource Integration Using a Large Knowledge Base in Carnot," *IEEE Computer*, Vol. 24, No. 12, December 1991, pp. 55-62.

[FIN93] T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzson, D. McKay, J. McGuire, P. Pelavin, S. Shapiro, and C. Beck. "Specification of the KQML Agent-Communication Language," Enterprise Integration Technologies, Palo Alto, CA, Technical Report EIT TR 92-04, updated July 1993.

[FLO95] D. Florescu, L. Raschid and P. Valduriez, "Using Heterogeneous Equivalences for Query Rewriting in Multidatabase Systems," *Proceedings of the International Conference on Cooperative Information Systems* (CoopIS95), Vienna, Austria, May 9-12, 1995.

[FLO96] D. Florescu, L. Raschid and P. Valduriez, "A Methodology for Query Reformulation in CIS using Semantic Knowledge," *International Journal of Intelligent and Cooperative Information Systems*, special issue on Formal Methods in Cooperative Information Systems, 1996.

[GEN94] M. R. Genesereth and S. P. Ketchpel, "Software Agents," *Communications of the ACM*, Vol 37, No. 7, July 1994, pp. 49-53.

[GOH94] C. H. Goh, S. Madnick, and M. Siegel, "Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment," *Proceedings of the Third International Conference on Information and Knowledge Management*, November, 1994, pp. 337-346.

[HAM93] J. Hammer and D. Mcleod "An Approach to Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems," *International Journal of Intelligent and Cooperative Information Systems*, Vol.2, No. 1, 1993, pp.51-83.

[ISO92] ISO Technical Committee 184, Subcommittee 4, "Product Data Representation and Exchange - Part 11: The EXPRESS Language Reference Manual," *ISO Document*, ISO DIS 10303-11, August 1992.

[KAM94] N. Kamel, P. Wu, and S. Y. W. Su, "A Pattern-Based Object Calculus," *International Journal on Very Large Data Bases*, The Boxwood Press, Pacific Grove, CA, Vol. 3, No. 1, January 1994, pp. 53-76.

[KIM91] W. Kim and J. Seo, "Classifying Schematic and Data Heterogeneity in Multidatabase Systems," *IEEE Computer*, Vol. 24, No. 12, December 1991, pp 12-18.

[LAN92] S. Lander and V. Lesser, "Customizing Distributed Search Among Agents with Heterogeneous Knowledge," *Proceedings of the First International Conference on Information and Knowledge Management*, Baltimore, MD, November 1992, pp. 335-344.

[LAN93] S. Lander and V. Lesser, "Understanding the Role of Negotiation in Distributed Search Among Heterogeneous Agents," *Proceedings of the International Joint Conference on Artificial Intelligence*, Chambery, France, August/September 1993, pp. 438-444.

[LU95] J. Lu, G. Moerkotte, J. Schue, and V.S. Subrahmanian, "Efficient Maintenance of Materialized Mediated Views," *Proceedings of 1995 ACM SIGMOD Conference on Management of Data*, San Jose, CA, May 1995.

[MOE92] T. Moehlman, V. Lesser and B. Buteau, "Decentralized Negotiation: An Approach to the Distributed Planning Problem," Group Decision and Negotiation 1:2, K. Sycara (ed.), Kluwer Academic Publishers, Norwell, MA, 1992, pp. 161-192.

[ODM93] ODMG Committee, "The Object Database Standard - ODMG 93", Morgan Kaufmann, CA 1993.

[OMG91] OMG Committee, "The Common Object Request Broker: Architecture and Specification," *OMG Document*, Revision 1.1, No. 91.12.1, December 1991.

[PAP96] Y. Papakonstantinou, H. Garcia-Molina and J. Ullman, "Medmaker: A Mediation System Based on Declarative Specifications." *Proceedings of International Conference on Data Engineering*, New Orleans, USA, February 1996, pp. 132-141.

[QIA93] X. Qian, "Semantic Interoperation Via Intelligent Mediation*," *Proceedings of 1995 International Workshop on Research Issues in Data Engineering*, Vienna, Austria, April 1993, pp. 228-231.

[QIA95] X. Qian and L. Raschid, "Query Interoperation Among Object-Oriented and Relational Databases," *Proceedings of 1995 International Conference on Data Engineering*, March 1995, Taipei, Taiwan, pp. 271-278.

[SAU96] Gunter Sauter and Wolfgang Kafer, "BRIITY - A Mapping Language Bridging Heterogeneity," *Proc. of the ITG/GI/GMA Conf., Software Technology in Automation and Communication* (STAK), Munchen, Germany, March 1996.

[SHY96] Y. M. Shyy, J. Arroyo-Figueroa, S. Y. W. Su, and H. Lam, "The Design and Implementation of K: A High-level Knowledge Base Programming Language of OSAM*.KBMS", *VLDB Journal*, Vol. 5, No. 3, 1996, pp. 181-195.

[SOM93] SOMobjects Developer Toolkit Version 2.0 Development Committee, "SOMobjects Developer Toolkit Users Guide", IBM, Version 2.0., 1993.

[SU89] S. Y. W. Su, V. Krishnamurthy and H. Lam, "An Object Oriented Semantic Association Model (OSAM*)," Chapter 17 in *Artificial Intelligence: Manufacturing Theory and Practice*, Institute of Industrial Engineers, Industrial Engineering and Management Press, Norcross, GA, 1989, pp. 463-494.

[SU91] S. Y. W. Su and J. H. Park, "An Integrated System for Knowledge Sharing among Heterogeneous Knowledge Derivation Systems," *International Journal of Applied Intelligence*, 1, 1991, pp. 223-245.

[SU93] S. Y. W. Su, M. Guo and H. Lam, "Association Algebra: A Mathematical Foundation for Object-Oriented Databases," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 5, Oct. 1993, pp. 775-798.

[SU95] S. Y. W. Su, H. Lam, T. F. Yu, et al, "An Extensible Knowledge Base Management System for Supporting Rule-based Interoperability among Heterogeneous Systems", *Conference on Information and Knowledge Management (CIKM)*, Baltimore, MD, November 28 - December 2, 1995, pp. 1-10.

[SU96] S. Y. W. Su, H. Lam, T. F. Yu, et al, "NCL: A Common Language for Achieving Rule-Based Interoperability among Heterogeneous Systems," *Journal of Intelligent Information Systems (JIIS)*, Special Issue on Intelligent Integration of Information, 1996, pp. 171-198.

[VEN91] V. Ventrone and Sandra Heiler, "Semantic Heterogeneity as a Result of Domain Evolution," *SIGMOD Record*, Vol. 20, No. 4, December 1991, pp 16-20.

[WIE91] G. Wiederhold, "Obtaining Information from Heterogeneous Systems," *Proceedings of the First Workshop on Information Technologies and Systems (WITS'91)*, Cambridge, MA, MIT Sloan School of Management, December 14-15, 1991, pp. 1-8.

[WIE92] G. Wiederhold, "Mediators in the Architecture of Future Information Systems," *IEEE Computer*, Vol. 25, No. 3, March 1992, pp. 38-49.

[WIE94] G. Wiederhold, "Interoperation, Mediation, and Ontologies," *Proceedings of the International Symposium on the Fifth Generation Computer Systems (FGCS94)*, Workshop on Heterogeneous Cooperative Knowledge-Bases, Vol. W3, ICOT, Tokyo, Japan, December 1994, pp.33-48.

[WIE95] G. Wiederhold and M. Genesereth, "The Basis for Mediation," *Proceedings of the International Conference on Cooperative Information Systems* (CoopIS95), Vienna, Austria, May 1995, pp. 138-155.

BIOGRAPHICAL SKETCH

Tsae-Feng Yu was born on August 6, 1964, in Taichung, Taiwan, Republic of China. He received his Bachelor of Education degree in industrial arts education from National Taiwan Normal University, Taiwan, Republic of China, in 1987. He received his Master of Computer Science degree from the University of Texas at Arlington in 1992. Prior to going abroad for graduate study, he was a teacher of industrial technology education in Yung-Ho Junior High School, Taipei, Taiwan. He will receive his Doctor of Philosophy degree in computer and information science and engineering from the University of Florida, Gainesville, Florida, in May 1997. His research interests include object-oriented information modeling, heterogeneous and distributed database systems, distributed query processing and mediation process.